Queensland University of Technology

SELinux Policy Management Framework for HIS

by
Luis Franco Marin (05592763)
BSc., MIT

This thesis is presented in fulfilment of the
requirements of the degree of
Master of Information Technology (Research)

Faculty of Information Technology

Principal Supervisor:
Dr. Tony Sahama

Associate Supervisor:
Prof. Bill Caelli

2008

I. Certificate of Original Authorship

The work contained in this thesis has not been previously submitted to meet requirements for an award at this or any other higher educations institution. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made.

Luis Franco Marin

II. Abstract

Health Information Systems (HIS) make extensive use of Information and Communication Technologies (ICT). The use of ICT aids in improving the quality and efficiency of healthcare services by making healthcare information available at the point of care (Goldstein, Groen, Ponkshe, and Wine, 2007). The increasing availability of healthcare data presents security and privacy issues which have not yet been fully addressed (Liu, Caelli, May, and Croll, 2008a).

Healthcare organisations have to comply with the security and privacy requirements stated in laws, regulations and ethical standards, while managing healthcare information. Protecting the security and privacy of healthcare information is a very complex task (Liu, May, Caelli and Croll, 2008b). In order to simplify the complexity of providing security and privacy in HIS, appropriate information security services and mechanisms have to be implemented. Solutions at the application layer have already been implemented in HIS such as those existing in healthcare web services (Weaver et al., 2003). In addition, Discretionary Access Control (DAC) is the most commonly implemented access control model to restrict access to resources at the OS layer (Liu, Caelli, May, Croll and Henricksen, 2007a). Nevertheless, the combination of application security mechanisms and DAC at the OS layer has been stated to be insufficient in satisfying security requirements in computer systems (Loscocco et al., 1998).

This thesis investigates the feasibility of implementing Security Enhanced Linux (SELinux) to enforce a Role-Based Access Control (RBAC) policy to help protect resources at the Operating System (OS) layer. SELinux provides Mandatory Access Control (MAC) mechanisms at the OS layer. These mechanisms can contain the damage from compromised applications and restrict access to resources according to the security policy implemented.

The main contribution of this research is to provide a modern framework to implement and manage SELinux in HIS. The proposed framework introduces SELinux Profiles to restrict access permissions over the system resources to authorised users. The feasibility of using SELinux profiles in HIS was demonstrated through the creation of a prototype, which was submitted to various attack scenarios. The prototype was also subjected to testing during emergency scenarios, where changes to the security policies had to be made on the spot. Attack scenarios were based on vulnerabilities common at the application layer.

SELinux demonstrated that it could effectively contain attacks at the application layer and provide adequate flexibility during emergency situations. However, even with the use of current tools, the development of SELinux policies can be very complex. Further research has to be made in order to simplify the management of SELinux policies and access permissions. In addition, SELinux related technologies, such as the Policy Management Server by Tresys Technologies, need to be researched in order to provide solutions at different layers of protection.

III. Keywords

Information Security, Security for Health Systems, Health Information Systems, Access Control, Security Enhanced Linux, Mandatory Access Control, Operating Systems, Trusted Systems.

IV. Acknowledgements

- My special thanks are to my principal supervisor, Dr. Tony Sahama, for his support and guidance through my research project. Also, I would like to thank him for his time and help that was never denied.

- I would like to thank my associate supervisor, Prof. Bill Caelli, for helping me in giving direction to my research and for sharing his vast knowledge with me.

- I would like to thank Prof. Peter Croll for his guidance during the early stages of my research.

- I wish to express my gratitude to Miss. Vicky Liu for sharing her knowledge and experiences with me, but mainly for her friendship.

Finally, I would like to acknowledge the Queensland University of Technology (QUT) for the financial support provided for the development of this research. Thanks to the scholarships awarded to me (QUT Masters Scholarship and Fee Waiver), I was able to successfully finish this research.

V. Dedication

I would like to dedicate this thesis to:

- My parents, Luis Franco Serrano and Graciela Marin Chavez, for their unconditional guidance and support during my whole life. Also, for teaching me the values and principles that have ruled my life.
- My sister, Lizzy Franco Marin, for her kindness, guidance, and advice when I needed it most.
- My partner, Leesa Johnson, for her support and help during the development of my professional career in Australia. Also, for being the pillar in my search for success.
- My friends, for encouraging me when I needed it most.

VI. Preface

Chapter 4 in this thesis is based on Franco, Sahama, and Croll (2008).

VII. Table of Contents

VIII. List of Figures

IX. List of Tables

Chapter 1

Introduction

1 Introduction

Healthcare organisations make extensive use of ICT to improve the quality of healthcare services (Liu, May, Caelli, and Croll, 2007b). The use of ICT along with the ability to collaborate between organisations can help to reduce errors and costs in existing healthcare services (Goldstein et al., 2007). Different technologies such as Electronic Health Records (EHRs) are used to support patients while delivering healthcare services at the point of care. The increasing availability of healthcare information presents security and privacy issues which have not yet been fully addressed (Liu et al., 2008a).

Healthcare organisations have to mitigate security and privacy threats in order to comply with laws, regulations and ethical standards. Unmitigated security risks could have serious consequences such as penalties, prosecution, loss of credibility and so on. In addition, once healthcare information is compromised the damage cannot be undone.

Protecting the security and privacy in HIS is a very complex task (Liu et al., 2008b). In order to simplify the complexity while providing security and privacy in HIS, appropriate information security services and mechanisms have to be implemented. Encryption is a security mechanism which is widely used in HIS (Weaver et al., 2003). However, this mechanism does not protect integrity and availability, and does not restrict privileges over the systems' resources.

Access control is a fundamental security mechanism in multi-user and resource-sharing computer systems such as HIS (Liu et al, 2007a). Authentication and authorisation mechanisms need to be provided in a computer system in order to prevent unauthorised disclosure or modification of information. Authentication mechanisms are the basis for verifying the identity of users. Nevertheless, authorisation is an indispensable component in order to inspect and approve users' access rights over the information and resources in the system (Weaver et al., 2003).

It is important to provide appropriate access control mechanisms at the OS layer due to the level of granularity that can be applied while restricting access to resources. The majority of current information systems, including HIS, are constructed based on OSs that use DAC mechanisms to restrict access to resources (Liu et al., 2008a). However, OSs that enforce a DAC model are not very effective in protecting the resources that they manage (Loscocco et al, 1998).

In this thesis, SELinux is proposed as a modern approach that could be used in HIS to implement MAC mechanisms at the OS layer. In 2000, the National Security Agency

(NSA) released SELinux as a flexible MAC implemented in the Linux kernel. However, it was not until 2003, that SELinux was included in the mainline Linux kernel 2.6. Before SELinux was released, access to OSs that enforce a MAC policy was very complicated because they were classified or expensive. SELinux is not only available to the general public, but it can also be obtained for free (enabled by default since Fedora Core 4). This allowed researchers around the world to explore potential applications in which MAC at the OS layer could be used in different types of environments and scenarios.

SELinux can be seen as part of the Open Solutions specified by Goldstein et al. (2007) which are the keys to achieving better healthcare systems. Linux is the main representative Open-Source Software (OSS) solution which is currently used in organisations around the world due to its reliability and security. Linux is the main OS that supports Web Servers, Printer servers and so on. However, it is predicted that in the future, Linux will also play an important role in supporting mission-critical applications (Weiss, 2008).

SELinux is not only an interesting proposition to enforce MAC at the OS layer due to its public and free availability and extensive use of OSS, but also due to the flexible and fine-grained features it provides. SELinux was created based on a microkernel architecture called the Flask architecture, which provides flexibility while supporting a variety of security policies in diverse organisational environments. Also, the Flask architecture allows SELinux to provide security at the application layer through the use of user-space object managers and an experimental policy management server.

SELinux implements a flexible and granular MAC called Type Enforcement (TE) and a type or RBAC built upon TE. In this thesis, TE and RBAC in SELinux are proposed as the preferred solution over systems that implement Multi Level Security (MLS). OS that implement MLS such as those based on the Bell-LaPadula or Biba model, are considered to be inflexible and do not fulfil security requirements for commercial organisations (Clark and Wilson, 1987). RBAC and TE in SELinux protect the confidentiality, integrity and availability of the resources along with the enforcement of the least privilege principle and domain separation. TE in SELinux provides a high level of granularity and flexibility while protecting the resources in the system. TE can be used to create sandboxes to restrict the resources which processes can access. RBAC helps to minimise the complexity of the user management task in large scale systems and to enforce the least privilege principle.

SELinux has already been proposed as a fundamental component of trusted architectures to protect resources in HIS (Henricksen, Caelli, and Croll, 2007; Liu et al., 2007a). However, this research is based on early versions of SELinux and need to be updated. SELinux is a technology that has changed dramatically since its release in 2003 because of the increasing interest in this technology. Tools and related technologies are constantly being developed to address complexities in the use of SELinux, making it more feasible for commercial purposes. In this thesis, a modern framework, based on SELinux Profiles, is proposed to aid in the implementation and management of SELinux in HIS.

SELinux Profiles are introduced to demonstrate how TE and RBAC can be used to protect resources in the system. SELinux profiles are defined as the authorised environment for subjects which determine the way authorised users interact with subjects and objects 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. Along with TE and RBAC, 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.

Finally, this thesis is intended to demonstrate that application layer security alone is not enough to satisfy security requirements in HIS. In HIS, as in any other computer system, security has to be properly implemented in four layers of security, which are: application layer, OS layer, hardware layer and network layer. Mechanisms at the application layer can provide a high level of granularity while protecting sensitive information so that complex security policies can be satisfied. The OS is the one who mediates accesses from applications to the hardware components. Therefore, access control mechanisms at the OS layer can provide a high level of granularity protecting processes, sockets, files, etc. The hardware layer can provide assurance that the OS booted as expected and can provide cryptographic keys for encryption or authentication. Finally, network layer security has to be provided in order to prevent malicious traffic from interacting with the system.

Currently, the most common way to protect access to information is through the use of security mechanisms at the application layer and DAC mechanisms at the OS layer. Applications such as web servers and databases (DBs) have been enhanced with access control mechanisms, but there are still vulnerabilities that have been exploited by attackers in recent years. Viruses, Worms and Trojans are used to attach backdoors in legitimate applications so that they behave in an unexpected manner. In an OS which implements DAC mechanisms to restrict access to resources, the attacker could use a compromised application to interfere with other processes or access restricted resources. Also, an attacker could use the privileges obtained from the compromised application to escalate privileges and compromise the whole system.

In this thesis, the aim is to demonstrate that SELinux is a viable solution at the OS layer, helping to satisfy security requirements in HIS. The main contribution is to provide a modern framework to implement and manage SELinux in HIS. The proposed framework introduces SELinux Profiles to restrict access permissions over the system resources to authorised users. SELinux Profiles are introduced to demonstrate the flexibility and granularity of SELinux MAC mechanisms. These mechanisms can be used to create sandboxes to contain damage from compromised applications.

Chapter 2

Background

Contents:

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. This was followed by initiatives from the Department of Defence (DoD) in the USA, to create an access control model which could protect the confidentiality and integrity of information at different levels of sensitivity. During the 1980s it was clear that MLS was not a proper solution for commercial environments and effort was spent in the creation of more suitable access control models. 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 in the early 1960s with a 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 enforce a MAC security policy has made an important change in the arena of Trusted OSs. Systems like Trusted Solaris, TrustedBSD 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.

This chapter introduces the concept of access control and its evolution from the early 1960s to today. The chapter is composed of sections which define and describe the main terms, technologies and concepts related with access control.

2.2 CIA principles

The term CIA corresponds to the three core principles of information security in computer systems, which are:

- **Confidentiality.** Security principle which refers to limiting information access and disclosure to authorised users while preventing access or disclosure to unauthorised users.
- **Integrity.** This Security principle refers to the protection of information against unauthorized modification or deletion.

- **Availability.** It is defined as the security principle which assures that the information is always accessible for authorised users.

These principles in a computer system have to be preserved in order to protect the security and privacy of the information while it is stored, transmitted or processed. Access control is a critical element in computer systems in order to preserve any of these principles. The use of access control aids to minimise threats leading to unauthorised disclosure, modification, or deletion of the information. In order to determine if an access to the information is authorised or denied, access control mechanisms are used in computer systems. It is very clear how access control could assist in protecting the confidentiality and integrity in a computer system through the authorisation or denial of access permissions to specific users. However, it is not clear on how access control mechanisms protect the principle of availability. In order to exemplify how access control can protect the availability of the information let us assume an attacker was able to access restricted information and delete it making it unavailable to authorised users.

2.3 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 organisations implement a type of access control to restrict access to facilities or information.

2.3.1 Authentication and Authorisation

Both, authentication and authorisation are fundamental to access control. Authentication is about verifying the identity of a user, while authorisation 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 authorisation. Authorisation 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 authorised access permissions on the resources. The token used to perform authentication is called an authenticator. Authenticators can be categorised as follows:

- **Knowledge Based.** Something that the user knows, characterised by secrecy and obscurity. A common example is the use of passwords.
- **Object-Based.** Something that the user has, characterised by physical possession of a token. Examples of these authenticators are the RSA secure identification tokens.
- **ID-Based**. Something that the user is, characterised by uniqueness to a person. Examples of these authenticators are a passport or biometrics.
- **Location-Based.** Somewhere the user is, characterised by location. These authenticators might involve position and tracking technologies. Examples include systems that make use of global positioning systems (GPS).

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 authorisation, while verifying the identity of the user during the log in process. Once in the system, users have to be restricted to authorised operations in order to preserve the three security principles. Authorisation 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, authorisation is the preferred solution to determine access permissions that users have while in the system according to the security policy implemented in the system.

2.3.2 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 categorised 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).

2.3.2.1 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 Defence [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 authorised 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 recognised 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.

2.3.2.2 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. (1998), 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 organisation.

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. As stated in the TCSEC (DoD, 1985), 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.

2.4 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 authorised to read, write and execute the contents of the directory /healthcare/doctor.

In OSs which makes use of ACLs to manage access rights, each object has a security attribute that identifies its ACL. This list has an entry for each system user with access privileges. When a subject performs and 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.

2.5 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 authorised 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 authorise 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.

The reference monitor should comply with the following design principles:

- **Tamper-proof.** The reference monitor should be designed in such a way that it is impossible for an attacker to interfere with the reference validation mechanisms and consequently affecting the access checks.
- **Completeness.** The reference validation mechanisms must always be invoked.
- **Verifiable.** The reference validation mechanisms must be small and simple as possible so that it can be subject to analysis and testing.

The level in which a system complies with these design principles provides the confidence in the correctness of the IT systems' secure mechanisms.

Because the reference monitor does not support a particular policy or a particular implementation, it has been used as the basis for the design, development and implementation of highly secure IT systems for over three decades. The first implementations of the reference monitor in computer systems were known as security kernels. Security kernels are the combination of hardware and software of an OS which are responsible for the enforcement of a security policy. In the subsequent years some form of the security kernel has been implemented on most of the OSs that we know. For example, in traditional Linux systems the access control database is hard-coded in the kernel and checks are done comparing user identifiers for subjects against access permission modes for objects.

The introduction of the reference monitor in off-the-shelf systems does not satisfy completely the organisation requirements (Ferraiolo et al., 2003). Even if the systems follow all the design principles of the reference monitor, it is not sufficient. This is because the reference monitor does not specify any particular access control policy, it just provide the assurance that the system is secure. Consequently, organisations have to be attained to the access control policies hard-coded by the vendor. In order for a system to comply with all the requirements of a complete access control system it has to be flexible, manageable, and scalable along with the design principles of the reference monitor (Ferraiolo et al., 2003).

2.6 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 behaviour. The U.S.A. DoD was responsible for funding most of these initiatives in order to protect classified military and strategic information against compromise during storage or processing. 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 organisations. 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.

2.6.1 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. It is based on mathematical models in which secure states are used to provide confidentiality of the information (LaPadula, 1996). A secure state can be defined as the condition in which no unauthorised 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 labelled 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.

In the Bell-LaPadula model, access decisions for subjects over objects in the system are restricted by two MAC security properties and one DAC security property, which are (McLean, 1985):

- **Simple Security Property.** This property is also known as the “no-read-up” property. The property is satisfied if for every read access from a subject to an object, the subject’s security attributes dominate the object’s security attributes.
- **Star Property.** This property is also known as the “no-write-down” property. The property is satisfied if for every write access from a subject to an object, the object’s security attributes dominate the subject’s security attributes.
- **Discretionary Security Property.** This is considered to be a DAC property not because the access to the information is on discretion of the owners of the information, but because it makes use of an Access Matrix. This property is satisfied when the entry for the subject and object can be found in the Access Matrix.

By following these properties the Bell-LaPadula model ensures that the security state is preserved, thus the confidentiality of the information. It is important to note that these principles can prevent attacks from Trojans and preserve the confidentiality of the information. However, it cannot prevent unauthorised modification of the information. For example, if a subject with a security clearance of *Top Secret* is tricked to run a compromised application, this application cannot copy anything from the *Top Secret* to any lower level, but it can delete all the information.

2.6.2 Biba Model

In order to solve the problem of unauthorised 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. These principles are defined as follows:

- **Simple Security Property.** This property is satisfied if for every read access from a subject to an object, the object's security level dominates the subject's security level.
- **Integrity Star Property.** The property is satisfied if for every write access from a subject to an object, the subject's security level dominates the object's security level.

By following these principles the integrity of the information is preserved by restricting the way in which the information flows from higher to lower integrity levels. However, as in the Bell-LaPadula model, integrity is protected while confidentiality is sacrificed.

2.6.3 Clark-Wilson Model

In 1987, David Clark and David Wilson published a paper in which they discuss that existent MLS models such as the Bell-LaPadula and Biba models are unsuitable for commercial organisations (Clark, and Wilson, 1987). They stated that these models are created for environments in which the confidentiality of the information is more important than the integrity such as military environments. On the other hand, for commercial organisations the integrity of the information is of primary concern. Integrity in commercial organisation is reflected on the accuracy and authenticity of the information while allowing only authorised subjects to modify objects in the computer system.

The Clark-Wilson model provides an integrity framework which attempts to follow controls used in bookkeeping and auditing through certification and enforcement. Certification provides proof that only legitimate transactions occur and to keep record in case an illegal transaction occur and enforcement provides separation of responsibilities. The certification and enforcement rules define the data items and procedures that provide the basis for the integrity policy.

The Clark-Wilson model proposed two principles to ensure the integrity of the information and protect against fraud and error: well-formed transactions and separation of duties (SoD). Well-formed transactions are the operations that make the system transit from one state to another keeping the consistency while restricting the way in which users can modify the data. SoD is used to prevent a single user manipulating data illegally or penetrating the security system. SoD is achieved when the one who certifies the transaction is a different entity from the one who implements it.

The basic unit of access control in the Clark-Wilson model is the “access control triple” composed by user, Transformation Procedure (TP) and Constrained Data Item (CDI). The entities defined by the Clark-Wilson model are (Clark, and Wilson, 1987):

- **CDI.** The data items within the system in which integrity must be preserved.
- **TP.** TPs can be seen as transactions whose main purpose is to change a set of CDI from one valid state to another.
- **Integrity Verification Procedures (IVP).** IVP are responsible to confirm that a CDI is in a valid state according to the integrity specifications.
- **Unconstrained Data Item (UDI).** These are the data items not covered by the integrity model, subject only to discretionary controls.

The model enforces MAC in the way that users are allowed to access only TPs and TPs are the only ones allowed to access CDIs. The system can ensure that only TPs manipulate CDIs, but it cannot ensure that the TPs perform a well-formed transaction. For this reason certification and enforcement rules are stated in the model. There are five certification rules and four enforcement rules which ensure the integrity of the data within the system, these are (Clark, and Wilson, 1987):

- **Certification Rule 1**. All the IVPs must properly ensure the correct state of all CDIs when the IVP is running.
- **Certification Rule 2.** Every TP that modifies a CDI must be certified to do it in a valid way.
- **Certification Rule 3.** In order to achieve SoD the list of user authorised to access TPs must be certified.
- **Certification Rule 4.** All TPs must be certified to write to an append-only CDI (a log) all information necessary to reconstruct the operation.
- **Certification Rule 5**. Any TP that takes a UDI as input must be certified to convert the UDI into CDI in a valid way.
- **Enforcement Rule 1.** CDIs can only be changed by certified TPs.
- **Enforcement Rule 2**. A user can access only the TPs for which it is authorised.

- **Enforcement Rule 3.** Every user must be authenticated by the system before accessing any TP.
- **Enforcement Rule 4**. The security officer or security administrator is the only one allowed to authorise users for TPs.

One of the main differences between MLS models and the Clark-Wilson model is that the Clark-Wilson model relies on application-level controls (Ferraiolo et al., 2003). In the military environment it is important to provide confidentiality which could be enforced with kernel-level controls restricting low-level read and write operations. In the commercial environment information has to be modified only by those authorised to do this task and in a controlled manner. This cannot be provided with kernel-level controls (Ferraiolo, 2003). Another important characteristic of the model is the SoD which is achieved through the segregation of the operation on the organisations. Operations have to be subdivided in small sub processes which can be achieved by different individuals in order to avoid fraud and error.

2.6.4 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 organisation. 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 organisational 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 authorisation.** A subject's active role must be authorised for the subject.
- **Transaction authorisation.** A subject can complete a transaction only if the transaction is authorised for the subject's active role.

In a subsequent paper (Ferraiolo, Cugini, and Kuhn, 1995) the term transaction is changed for operation which represents the access permissions to a set of objects. Also, this paper makes a clear distinction between users and subjects in a computer system. Here, subjects are defined as active entities in the system which performs operations on behalf of users. In computer systems, subjects are the ones who are assigned a set of active roles for which the user must be authorised.

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. RBAC0 is the base model including the minimal requirements for a RBAC system. RBAC1 and RBAC2 include RBAC0, 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 organisations. A simplified implementation of RBAC in a commercial organisation could use RBAC0 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 initiative resulted in a standard for RBAC which was submitted in 2002 to the international standard process, and has been the basis for the creation of products that conform to the RBAC model. This standard was based on the RBAC96, but it adds features in regard of requirements from the commercial vendor community.

RBAC0 (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. In the RBAC standard, the role hierarchy supports two different types of inheritance: permissions are inherited upwards and the set of roles available to a user is aggregated downwards. For example, if role R is in a higher hierarchy than role R1 then any permission assigned to R1 is implicitly assigned to R, and any user assigned to R can activate R1.

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. Depending on the rules enforced, a user in one role is restricted from being a member of one or more other roles. Dynamic SoD also limits the permissions available to a user by placing constraints on the roles that can be activated within or across a user's session.

RBAC has been implemented in a wide variety of areas, such as healthcare in which RBAC has a natural application (Reid, Cheong, Henricksen, and Smith, 2003). One of the reasons for its widespread adoption is the use of roles that facilitates the assignment and removal of privileges, and also allows the efficient analysis of the users' privileges authorisation. The possibility of audit the consistency of privilege allocation efficiently helps to reduce configuration errors which are very common in large scale systems such as healthcare systems. In addition, the assignment of users to roles allows enforcing powerful rules regarding conflict of interest and cardinality rules for roles.

In contrast with traditional ACL approaches such as traditional DAC systems, RBAC does not allows users to be directly associated with privileges, privileges are held by roles. In ACL systems, all the resources have a list of authorised users. This simplifies the task of finding subjects that can access an object, but it is very complicated to find the objects to which the subjects are allowed to access. In the case of systems that implement groups, once the group permissions are removed from an object, the user could maintain access permissions to the object. This happens because users access the objects using the user identifier (UID) or group identifier (GID). Even if the group permission is removed the user permission could stay without being noticed. RBAC helps to mitigate this risk since all accesses depend on the roles.

2.6.5 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 Haghighat, 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 traditional TE model is a table-oriented access control model which makes use of two tables for access control decisions. A global table called the Domain Definition Table (DDT) represents the domain-type group, that is, permissions between domains and types. Each column in the DDT table is a type and each row is a domain. A second table called the Domain Interaction Table (DIT) represents the domain-domain group, that is, permissions between domains. If a subject A attempts to access an object B, this access mode is referenced in the DDT table in the cell corresponding to the domain of A and the type of B. If the cell contains the requested access mode (read, write, or execute) then the permission is allowed, otherwise it is denied. In case a subject A attempts to access another subject B, the access permission is referenced in the DIT table in the cell corresponding for the domain of the subject A and the domain of the subject B. If the access permission (kill) exists in the cell, the access is allowed, otherwise it is denied.

There are two primary techniques that distinguish DTE from traditional TE which are (Badger et al., 1995):

- A high level policy specification language. The use of a high level language allows the reusability of access control configurations.
- DTE file security attributes are maintained implicitly. This allows simplifying the secure configuration establishments and removes the need to physically store a type label with every file.

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).

2.7 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.

Providing the appropriate security mechanisms in each of these layers helps to reduce the damage in the system in case an attack occurs. If security mechanisms are properly implemented in each layer, if one of these fails others would be able to mitigate the damage. If no mechanisms are implemented in one of the layers, there would be an increase in the vulnerabilities that can be exploited by an attacker.

Nevertheless, as a design principle, security has to be provided in a holistic manner covering every aspect of a computer system. This includes the four mentioned layers of security plus the corresponding policies and procedures. Note that without appropriate security policies and procedures in place, the security mechanisms in these layers could not be properly implemented. For example, if strict authentication mechanisms are used but users still can choose weak passwords, the probabilities of an attack are still high.

This research is mainly focused on the OS layer. The OS manages hardware components such as the disk drive, the keyboard, network cards, the monitor and the printer. Therefore, the OS is responsible for the appropriate operation of the computer. If appropriate security mechanisms are implemented at the OS layer, damage from compromised applications can be mitigated. However, in order to get the required support from the OS, MAC mechanisms have to be enforced. The OS has to enforce an access control policy which restricts the way in which subjects may access objects in the system. If the OS implements DAC mechanisms, an attack may not be contained and an exploit of a flawed application could result in the attacker ability to control the whole system.

2.8 Operating Systems Mandatory Access Control

In 1998, Loscocco et al. (1998) publicised a paper in which they expressed their concern about the way in which computer systems were built relying on the security at the application layer. This paper was the basis for the understanding that in order to construct secure systems it is important to have the support from the underlying OS.

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 through one of the following common exploits (Dalton, and Choo, 2001):

- **Misuse of privileges to gain access to system resources.** If an application is running with special privileges, the attacker who compromised the application will be able to use those privileges for unintended purposes. This can be solved even in a DAC system, running applications as unprivileged users.
- **Subversion of application-enforced access control.** If an application which provides information in an authorised fashion is compromised, the attacker will get access to restricted information. For example, this exploit could happen in a web server which provides information according to certain access rules. In this web server even if access to the information is not granted the web server always has access to the protected information. Consequently, if compromised, the attacker has access to the restricted information bypassing access control rules supposedly enforced by the application.
- **Supply of bogus security decision-making information.** This happen when the attacker has access to an application in charge to provide security information to another application. In this case the attacker can provide bogus information to compromise the other application or to have access to restricted resources.
- **Illegitimate usage of unprotected system resources.** An attacker could have access to restricted resources or escalate privileges by compromising an application and using its privileges to access resources that usually would not be available. For example, the attacker could compromise the web server and use it to escalate privileges thus compromising the whole system. This can be

contained by implementing MAC in the OS thus restricting the privileges of the application.

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.

2.8.1 Multics

Multiplexing Information and Computing Service (Multics) is an OS developed in the late 60’s which represented an important influence while developing modern trusted OS. Multics was a mainframe timesharing OS which initial planning began in 1964 as a joint project between the Massachusetts Institute of Technology (MIT), General Electric (GE), and Bell Labs. Multics was implemented in a GE-645 mainframe due to its special hardware requirements. In the late 60s GE provided Multics as a time-sharing commercial product which was sold to more than 80 sites (amongst them the Ford and GM sites). In 1969, Multics was used in the MIT Laboratory for Computer Science to provide campus-wide services for all the academic and administrative users. The MIT Laboratory for Computer Science concluded its research in Multics in the late 1970s and shut down its Multics service in 1988. The last known Multics system belonging to the Canadian Department of National Defence was shut down on October 30, 2000.

Multics was conceived as a general purpose timesharing utility. The main idea behind time-sharing computing is to allow simultaneous access while giving illusion of ownership. This type of systems require to address some issues such as: response time, convenience of manipulating data and program files, ease of controlling process during execution and above all, protection of private files and isolation of independent processes. Multics introduced numerous features to provide high availability and security in time-sharing systems. Multics introduces major security features since services were shared between different users, which may not be trusted between each other. Multics provided major security features at the file level via access controls. The major innovations introduced by Multics were:

- **Virtual memory segmentation**. Multics implemented a single level store for data access, discarding distinction between files and process memory. Processes' memory consisted of segments which were mapped into real addresses and the OS was responsible to read and write on them.
- **High-level language implementation**. Multics was one of the first OS developed in a high level language. In 1964, Early PL/I (EPL), a subset dialect of PL/I, was chosen as the programming language.
- **Shared memory multiprocessor.** Multics CPUs share the same physical memory.
- **Relational database.** In June 1976, Multics released the first commercial relational database, the Multics Relational Data Store (MRDS).
- **Security.** Multics was the first OS to be designed as a secure system from the beginning. It was evaluated by the TCSEC receiving the first B2 security rating which was the only one for years.

Even if Multics was designed to be secure from the beginning, it was broken several times. Nevertheless, this contributed to improve the security of the system and create the basis for modern security engineering techniques. Multics access model was based on a hardware supported ring structure which was one of the major revolutionary concepts introduced by Multics. A ring is defined as a domain in which a process runs. Multics introduced the use of 8 numbered rings, from 0 to 7. These rings were designed to protect data and functionality from errors and malicious code. The privileges gradually increase from ring 7 to ring 0. Ring 0 corresponds to the kernel, ring 1 to the system services, ring 2 to the OS extensions, ring 3 to the utility programs such as DB, and ring 4 to 7 correspond to application programs. Accesses from one ring to another could be stated as: Processes at ring i have privileges of every ring j > i. This statement allows, for example, the kernel to access every resource in the system and restrict the application to compromise the resources. Modern CPU architectures include some form of rings such as the Intel x86 processors which have 4 privilege levels.

Multics is the predecessor for many other OSs such as the UNIX family of OSs. It provided the basis for the introduction of mechanisms that can restrict application to compromise the whole system. The introduction of rings is still used in these days by OS such as windows and UNIX, even if none of them make complete use of the rings provided by technologies such as Intel.

2.8.2 Solaris Trusted Extensions

Trusted Solaris 8 was released in 2000 by Sun Microsystems as a secure variation of the Solaris 8 OS (UNIX-based OS). This variation of Solaris 8 was offered to those customers who need extra security. Trusted Solaris 8 offered MAC for end users, RBAC for system administrators, fine grained rights profiles allowing administrators to give end users specific access to the resources in the system. Before SELinux and Solaris 10 were certified, Trusted Solaris 8 was the only OS to achieve the Evaluation Assurance Level 4 (EAL4) granted by the Common Criteria for Information Technology Security Evaluation (CC). In 2004, Trusted Solaris received security certification under the CC against the Control Access Protection Profile (CAPP), Labelled Security Protection Profile (LSPP), and the Role-Based Access Protection Profile (RBACPP). The fact that Trusted Solaris 8 received the EAL4 assured customers that it met global standards for evaluating IT security. Trusted Solaris 8 was very interesting for organisations getting more and more concerned about information security.

In 2005, Sun Microsystems released the new version of Solaris, Solaris 10. However, this time instead of releasing a variation of Solaris with extra security features, they included some security features existing in Trusted Solaris 8 in Solaris 10, such as fine-grained privileges. This helped Sun Microsystems to have a more stable version of the system. However, some of the MAC features of Trusted Solaris 8 still needed to be added without impacting those users without high security requirements. For this reason, Sun Microsystems created the Solaris Trusted Extensions which included all those security features that were in Trusted Solaris 8.

In 2006, Sun Microsystems introduced the Solaris Trusted Extensions as an optional set of security extensions to Solaris 10. Trusted Extensions are part of the OpenSolaris project as an initiative of Sun Microsystems to build a developer community around Solaris OS technology. Their functionality consists of a set of label-aware services that are derived from Trusted Solaris 8 and are based on new security features introduced in Solaris 10. Solaris Trusted Extensions enforce a MAC policy providing MLS on all aspects of the OS, including device access, file, networking, printing and window management services. Sensitivity labels are applied to all active and passive entities in the system and accesses are restricted by the relationship between the label of the object and that of the subject. Solaris Trusted Extensions allow customers of all types with high security requirements to use the labelling features to protect sensitive resources. Services provided by Solaris Trusted Extensions include:

- **Labelled Networking**. Since Trusted Solaris 8, remote hosts can be single level or multilevel. Labels are assigned to hosts based on their network or IP address. For the purpose of communicating with multilevel hosts, Trusted Extensions make use of the Commercial IP Security Option (CIPSO) which encapsulates a sensitivity label as an IP option.
- **Labelled Devices.** Each device on the system, such as terminals, disk drives, USB thumb drives, CD-ROMs, printers, audio devices and so on, could be assigned with a label range from which it may accepts requests.
- **Labelled Desktop**. Trusted Extensions enforces MLS through two labelled desktop interfaces, the trusted Common Desktop Environment (CDE) and the trusted Sun Java Desktop System (JDS). The trusted JDS is the first labelled environment based on the GNOME open source desktop standard.
- **Label-aware System Management.** The MAC policy applies to all aspects of the OS and even system administrators cannot violate the policy inadvertently.
- **Audit Trails.** Audit records include the labels of subjects and objects, and additional label-related events. It provides multilevel support for auditable events. The audit system is protected against processes with high privileges so that they cannot observe the audit trail nor tamper with any records.

An important feature of Trusted Extensions is the use RBAC mechanisms for the delegation of administration tasks and enforcement of SoD among administrators, security officers, auditors, and users. Users can assume one or more roles to which they have been assigned. There is a single dedicated workspace for each role. The user authenticates in the system and the window system creates a new administrative workspace for the role and starts another session. Roles are aimed at administrative tasks using predefined roles such as root, system administrator, security administrator, system operator, and primary administrator. For example, the system administrator role creates accounts and zones, while a security administrator assigns labels to them.

In 2007, the combination of Solaris 10 and Trusted Extensions was submitted by Sun Microsystems to be evaluated by the Common Criteria. Solaris 10 with Trusted Extensions was certified to properly meet security requirements from the LSPP, CAPP, and RBACPP at an EAL4+.

Policy configuration in Solaris does not require the creation of long, error prone security policy files. This is the result of creating Solaris Trusted Extensions as a feature of Solaris 10 OS. This is because the security features are already there. Labels already exist in the system, but if the trusted extensions are not enabled, all labels are equal (at the same sensitivity level). The kernel keeps a boolean in order to determine whether or not labelling comparison should be used in policy enforcement.

2.8.3 Security Enhanced Linux

SELinux was initially developed by the NSA based on the implementation of MAC in microkernel systems. In 1993 researchers from the NSA and the Secure Computing Corporation (SCC) worked on a project called Distributed Trusted Mach (DTMach) which was based on the design of an implementation of TE into the Mach microkernel. This project evolved into the Distributed Trusted Operating System (DTOS) project which produced a prototype to be used by universities for research purposes. Once this project was concluded, the NSA along with the SCC and the University of Utah's Flux project decided to transfer the DTOS security architecture into the Fluke research OS. In 1999, as an outcome from this project the Flux Advanced Security Kernel (Flask) architecture was created as an enhancement of the DTOS architecture to provide better support for dynamic security policies.

In 1999, the NSA decided to begin the implementation of the Flask architecture in the Linux kernel as an initiative to demonstrate its viability and gain support for its use. SELinux was released in December 2000 by the NSA and developed with cooperation from highly recognised security institutions such as NAI Labs, SCC, and MITRE. Following the initial release of SELinux, the Linux community got interested and an increasing research was commenced to enhance features in the Linux kernel to work with this technology. As part of this initiative, the Linux community realised that the standard kernel needed to be extended to provide more flexibility for security add-ons. As a result, 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. At the same time, the Linux community incorporated the LSM features into the mainline kernel 2.6. SELinux merged into the mainline kernel 2.6 on 8th of August 2003 when the NSA with great support from the open source community finished adapting SELinux to 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. Other Linux distributions such as Debian, SuSe, and

Gentoo can be enhanced with SELinux packages, but it can be complicated to make it work properly. 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 labelling 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.

One of the main benefits of SELinux is the flexible policy configuration, that is, the policies are not hard-coded into the kernel (Mayer, Macmillan, and Caplan, 2007, pp. 10). This makes SELinux suitable for any organisation's security policy since every access rules in the security policy can be configured in SELinux. This property is possible using configuration files which contain all the rules from the security policy to be enforced by the kernel. This configuration files are called SELinux policies. As defined by Mayer et al. (2007) a SELinux policy is "the collection of rules that determine allowed access for a system".

2.9 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 mechanisms, through authentication and authorisation, determine the identity of users and authorise access permissions for authenticated users. In this way, only authorised users are allowed to disclose, modify and access systems’ resources. In computer systems it is important to determine that resources are accessed in a controlled manner. For this reason, ACLs and the Reference Monitor were created in order to restrict accesses in multi-user systems. Also, access control models have been created over the years to provide ways to control the access of resources. TCSEC provided the definition for two main access control models, DAC and MAC. As stated by TCSEC, in DAC, access to resources is at the discretion of users, while in MAC access to resources is restricted by the system according to a security policy. MAC is based on a MLS model called the Bell-LaPadula model which protects the confidentiality of classified information. In order to complement the Bell-LaPadula model, the Biba model was created to protect the integrity of classified information. MLS models are considered to be unsuitable for commercial organisations. For this reason, access control models such as the Clark-Wilson model, RBAC, and DTE were created. RBAC is one of the most widespread adopted models in commercial organisations, including healthcare organisations.

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. Therefore, the damage from compromised applications can be contained and the system resources are protected. Multics was the first trusted OS created for multi-user systems and was the base for the development of current OSs. 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 organisations. Both technologies have their own advantages and disadvantages. However, SELinux is a promising prospect for future implementations of MAC at the OS layer. This is due to the flexibility and high granularity provided by the SELinux MAC mechanisms, and the fact that is part of the open source projects.

Chapter 3

Access Control Security in HIS

Contents:

3 Access Control Security in HIS

3.1 Introduction

Healthcare organisations make extensive use of ICT in order to improve the quality and safety of healthcare services. The healthcare industry, in countries like USA and Australia are embracing technologies in order to meet patients' need to access their healthcare information at the point of care. The increasing availability of patients' healthcare information increases the security threats to which the users and systems are targeted. Healthcare organisations have to comply with laws, regulations and ethical standards while protecting their customers' healthcare information. In order to simplify the complexity while protecting the security and privacy in HIS, appropriate information security services and mechanisms have to be implemented. Access control mechanisms are the preferred solution while protecting healthcare information, due to the need to protect the information against unauthorised disclosure or modification.

The first sections in this chapter describe technologies that have been introduced in HIS, along with security issues that have been found while using these technologies. A subsequent section describes the laws and regulations that affect organisations which manage the healthcare information of their customers. The security requirements for healthcare organisations will then be identified, along with existing security mechanisms. Access control is then introduced as an indispensable security mechanism in computer systems. The final section is dedicated to explaining why DAC mechanisms do not satisfy security requirements in HIS and to propose a preferable solution using MAC mechanisms.

3.2 Health Information Systems

Healthcare organisations make extensive use of ICT to store, process and transfer patients' information, improving the quality, safety and efficiency of patients' care. HIS are those information systems which apply ICT to support functions which affect healthcare services offered by healthcare professionals, non professionals, businesses or customers. Currently the most widely implemented HIS are those which manage EHRs. The use of EHRs has been demonstrated to considerably improve patients' care, allowing healthcare information to be promptly available to patients and healthcare providers (Noumeir and Chafik, 2005; Win, Susilo, and Mu, 2006).

HIS are moving from organisation-centred to customer-focused and patient-centred healthcare systems, in which the main focus is prevention and homecare instead of institutionalised care (Blobel, 2007). In order to achieve this goal, every patient needs to have their own secure and private EHR, which would be available when and where it is needed most, that is, at the point of care. HIS need the latest technological advantages combined with interoperability between healthcare providers (Goldstein et al., 2007). In addition, corresponding security countermeasures have to be in place to provide security and privacy to healthcare information. By doing this, HIS can provide the expected efficiency and high quality in a secure and confidential manner regardless of the time and location.

According to Goldstein et al. (2007), the use of HIS along with the ability to collaborate between organisations can address existing issues in healthcare services, such as:

- Medical Errors.
- Not diagnosing and treating with the latest medical knowledge.
- Poor vital health statistic performance.
- Partial, fragmented patient medical records.

3.3 Technologies and Security Issues

HIS are constituted by a variety of technologies which monitor, inform and decide in regards to patients' care. These technologies allow patients to be more active in their personal care. As a result, these technologies help to improve the quality of care while reducing unnecessary costs and medical errors. Technologies will provide a variety of benefits for the healthcare sector. However, there are a number of privacy and security issues which have to be addressed in order to receive full advantage of its implementation. These issues include (Meingast, Roosta, and Sastry, 2007):

- Security of data while in transfer
- Security of data while stored
- Amount of data stored
- Access control to the data
- Data analysis rights
- Governing policies and regulations
- Non-repudiation
- Accountability

The following sections in this chapter describe different technologies that support patients, while delivering health and medical services at the point of care. Also, this includes a brief description of the security issues involved while using these technologies.

This research is concerned about the protection of sensitive healthcare information against unauthorised disclosure or modification. In current HIS, patients' sensitive healthcare information usually is in the form of EHRs. For this reason, this research is focused on the protection of resources at the OS layer in HIS that store EHRs.

3.3.1 Electronic Health Records

In the past, patients' health records were kept in paper-based stored in file repositories internal to the healthcare organisation. The use of EHRs shifts paper-based health records into digital representations which are accessible electronically. EHRs would possibly be no longer stored in repositories inside the healthcare organisation, but in DBs in a central repository outside the organisation.

The use of EHRs helps to avoid legibility errors and misunderstandings, reducing operation costs and improving patients' care. In the USA, each year prescription errors kill around 7,000 patients and cost more than US$6 billion (Kingsbury, 2008). As stated by Glen Tullman, Chairman and CEO of Allscripts in the year 2005, "the advent of electronic health records will be as significant as the discovery of the penicillin" (Saporito, 2005).

EHRs are the basis of next-generation HIS, which depend on the prompt accessibility of patients' healthcare information. According to the survey developed by the American Hospital Association (AHA), in which approximately 31 percent of U.S.A hospitals participated, more than two thirds of hospitals had fully or partially implemented EHRs by 2006 (Anonymous, 2007). By 2009, at least three countries outside the USA (including Australia) will reach the point in which half of their healthcare costumers will be benefited by an operational EHR network (Hieb, Rishel, Edwards, and Kelly, 2005). Currently, there are many initiatives from different governments around the world to set up regional and national EHR programs. Examples of these programs are the Nationwide Health Information Network (NHIN) in the USA and HealthConnect in Australia.

The content of EHRs can vary from one healthcare organisation to another, but generally EHRs contain personal data relating to prescriptions, current mental and physical health, diagnostic reports, etc. In nationwide EHR systems, the information of millions or even tens of millions of patients is digitalised for further storage, processing and transference. Security and privacy concerns have arisen due to the extent and ease of disseminating healthcare information. In addition, the increasing number and diversity of individuals who desire to gain access have also created security and privacy concerns. The disclosure of EHRs could lead to serious consequences, such as personal embarrassment, refusal of prospective job opportunities, difficulties in obtaining or continuing insurance contracts and loans, or rejection from family or social groups. Privacy and security issues are considered to be the most important inhibitor of the acceptance and implementation of EHR systems (Ray and Wimalasiri, 2006). Security and privacy concerns have arisen from past experiences, such as:

- The theft of a senior employee's laptop in March 2008 from the National Heart, Lung and Blood Institute (NHLBI) in the USA, compromising the names, birthdays, medical record numbers and diagnoses of 2,500 patients (Jean-Francois, 2008).
- In the UK the Department of Health admitted in February 2008 that a total of 4,147 smartcards have been stolen or lost. National Health System (NHS) clinical staff use smartcards to access confidential EHRs (BBC News, 2008).
- The world privacy forum in 2006 said it had received 20,000 reports of medical identity theft in the last 15 years (Dixon, 2006). In this report medical identity theft is stated as the ultimate invasion of privacy, based on the theft and use of the EHRs of others. In Medical identity theft, patients' healthcare records can be modified leading to serious consequences such as an erroneous treatment of the patient. Gartner Research predicted one million cases of medical identity theft in the U.S.A. by 2008 (Lopez et al., 2006).
- In one of the largest data breaches in the state of Oregon in the USA, EHRs of 365,000 patients were compromised. A computer disk and digital tapes containing unencrypted EHRs were stolen from a Providence home service employee (Rojas-Burke, 2006).

The concern of using EHRs is the security and privacy of patients' sensitive healthcare information. Security measures are not in place while storing, processing and transmitting EHRs. The lack of strict access control measures could lead to unauthorised accesses to sensitive healthcare information. As reported by the board of the eHealth Vulnerability Reporting Program (eHVRP) in September 2007 (PRWeb, 2007), EHR systems can be easily hacked using standard tools and techniques. Attackers can gain control over the applications running in the system, and access sensitive healthcare information. Not only hackers commit unauthorised accesses, but also authorised users can take advantage of their privileges. This has been demonstrated in recently, when a former University of California, Los Angeles (UCLA) Medical Centre employee was sentenced to 10 years in prison for selling medical records of high-profile patients to a media outlet (CNN, 2008).

3.3.2 Implantable eCare

Implantable eCare encompasses devices which are implanted inside the human body. Advanced versions of these devices can communicate with monitoring devices outside the human body through the use of wireless technologies. Implantable eCare devices have evolved from artificial implants (i.e. knees) to assistive devices such as pacemakers, Implantable Cardioverter-Defibrillator (ICD), and implantable drug-pumps.

3.3.2.1 Implantable Cardioverter-Defibrillators

The ICD, also known as automatic internal cardiac defibrillator (AICD), is a device similar to a pacemaker. ICDs are placed under the skin close to the heart to monitor patients' heart rhythm. In case of anomaly, the ICD delivers a jolt of electricity according to the pre-programmed disease specifications. The patient's physician is the only one authorised to program the ICD according to the characteristics of the patient. If the ICD is not properly programmed, it could create a state of discomfort and anxiety in the patient, or even cause a cardiac arrest. ICDs are currently considered the standard treatment for patients with risk of cardiac arrest (MayoClinic, 2006).

In recent years, ICDs have been enhanced with computational and wireless capabilities increasing the potential of this technology. ICDs can be remotely checked, relieving the patients' need to repeatedly visit their physician. ICDs send information using a radiofrequency signal to bedside monitors. External monitors are used to send information regarding the patient's heart activity to the physician. Wireless capabilities not only allow the physicians to download data from the ICD, but also to send commands to the device. This functionality allows physicians to easily adjust the activities of the ICD as needed. In the USA approximately 100,000 patients have been implanted with these new devices (Jewell, 2008).

In March 2008, a study made in the USA led by Dr. Tadayoshi Kohno, Dr. Kevin Fu and Dr. William Maisel discovered that these devices can be hacked (Halperin et al., 2008). The signal between the ICD and the external monitors is not encrypted hence is prone to attacks. The study identified two ways in which an attacker can manipulate ICDs to cause harm: reprogram the ICD inducing an inadequate behaviour or inducing a potentially fatal shock; or retrieve personal information from the devices, such as name and Social Security Number (SSN). In order to prevent attacks, the study developed three “zero-power” defence mechanisms (Halperin et al., 2008): zero-power notification, zero-power authentication and sensible key exchange.

There have not yet been reports of any attacks exploiting vulnerabilities in ICDs. Currently, these devices receive signals over several feet, but in the future devices will work with greater distances. Increasing the distances also increases the potential for information to be intercepted on route.

3.3.3 NanoCare

NanoCare is based on the development of micro components inserted and applied inside the human body. NanoCare is similar to implantable eCare; however these devices are smaller and are more active components. In the future, these devices are going to be used to search and destroy for cancer cells inside the human body, or to replace damaged internal tissue.

3.3.3.1 VeriChip

VeriChip is a glass encapsulated Radio-Frequency Identification (RFID) microchip manufactured by VeriChip Corporation. This microchip is implanted inside people for authentication and access control purposes. VeriChip is the size of a grain of rice and once implanted in the individual it is imperceptible to the human eye. The microchip is implanted during a small surgery, performed under a local anaesthetic. It is commonly implanted over the right arm’s triceps. The microchip responds to a determined frequency by returning a sixteen-digit number which uniquely identifies individuals.

In healthcare organisations VeriChips, along with similar RFID tags have been used as access control mechanisms (Swedberg, 2008). Physicians can use the microchips to confirm patients’ identity and obtain related EHRs from an associated secure DB. VeriChip has been implanted in the U.S.A. in elderly patients with Alzheimer so that they can be recognized and identified in case they get lost (Swedberg, 2007). In August 2006, VeriChip Corporation made a sale of US$750,000 for the creation of a system for infant protection, installed in the Brampton Civil Hospital, Ontario. In the same year, VeriChip Corporation in partnership with Austco, implemented the infant protection system Hugs and MyCall emergency response system in some Australian hospitals (i.e. Gympie Hospital, in Gympie, Queensland).

A research study made by Jonathan Westhues demonstrates that VeriChips can be hacked (Westhues, 2006). According to this research, anyone with the appropriate scanning equipment can read the data from the RFID tag in less than a second. If the data inside the microchip is used for access control purposes, the attacker could have access to sensitive information and/or resources.

3.3.4 Smart eHomes

Smart eHomes are homes in which different devices monitor the healthcare needs of the owner of the house. Monitoring devices automatically send patients' health information to a remote healthcare provider for analysis or for emergency needs.

3.3.4.1 Telehealth Monitoring

Telehealth is the use of ICT to provide access to healthcare services and information across long distances (Nickelson, 1998). Telehealth Monitoring is the exchange of health data between patients at home and medical staff at the hospital for diagnosis and monitoring purposes. Telehealth devices such as asthma or blood glucose monitors are able to measure patients' vital signs while patients do their normal activities at home. Monitoring devices send measures via wireless or Bluetooth to a home unit. The home unit stores and evaluates the information for clinical review at a remote location. Communication with the remote location is kept via phone lines or other types of networks.

Telehealth monitoring makes extensive use of wireless technologies for the transfer of patients' vital signs. Eavesdropping and skimming are common threats in wireless communication. Wireless networking increases the vulnerabilities of security breaches. Technologies such as Bluetooth and HomeRF introduce advantages in mobility and are easy to use. However, these technologies also bring security risks which could lead to the compromise of the information transmitted.

Most of the time, the information exchanged between patients and medical staff is sent through the Internet. The Internet is recognised to be insecure by default, hence everything sent through the Internet is at risk of been compromised (Daswani et al., 2007). Sending the information through the Internet could lead to a number of attacks such as:

- Loss of confidentiality. If the information is not encrypted or the secure communication protocol is not well implemented, the attacker could gain access to the information.
- Loss of Integrity. An attacker could modify the content of the information sent between patient and medical staff.

- Masquerade. The attacker could steal authentication information of the patient and masquerade as the patient to access restricted information or resources.

3.3.5 eEHR

In the future EHRs are going to be available through the Internet using a variety of tools which enable the storage, management, and intelligent use of EHRs (Goldstein et al., 2007). The purpose of these technologies is to allow customers to access their information at the point of care 24/7.

3.3.5.1 Web Portals for EHRs

Due to the increasing use of EHRs, systems that allow access to EHRs at anytime and from anywhere are becoming more common. The idea is to make EHR accessible around the world at any time, like money through an ATM. There are companies which have seen the potential of these systems and have already started to create systems that prepare them for future healthcare markets.

WebPages like iHealthRecord.com, run by Medem Inc., allow patients to create their own medical records which they can access whenever they need to, totally free of charges. This portal, introduced in 2005, allows users to easily create, update, and access their healthcare records. The records can be shared with whomever they choose (physicians and other healthcare providers). This information can be accessed from wherever the patients need it and with the assurance that the security and privacy of the information will be protected. Currently, iHealthRecords.com states that 100,000 physicians in the USA use the system to access their patients' records.

Revolution Health, supported by AOL co-founder Steve Case, was founded in 2005 as a health-related portal and social network site. In this portal, patients are able to rate physicians and get advice from other people through online discussions. It also provides tools to calculate amounts of desired weight loss, and the amount of money they can save if they quit smoking. The most interesting feature is the Health Records Express service, which allows users to manage their healthcare records. In order to load healthcare records into the system, users have to request them from their healthcare provider. Once the records have been obtained, the user has to fax the records to Revolution Health. Within 24 hours, the user will be able to manage their healthcare records on-line.

In October of 2007, Microsoft (MS) released a web site for managing personal and healthcare information called MS Health Vault. MS Health Vault is a free site which can store medical histories, immunisations and other records from physicians. The site also includes search features which can be used to find certain treatments, or diseases. A major characteristic of Health Vault is that it can be connected to various devices, of which MS offers the device drivers. Actually, Windows Vista has a gadget allowing easy access to Health Vault. This will allow the easy transfer of data from other web tools and healthcare providers. Health Vault restricts the use of medical records to the owner of the records, any access or modification has to be authorised by the owner of the information.

In February 2008, Google announced an initiative to launch an online DB, where patients can easily store and maintain health information. Google Health will help to share EHRs between healthcare organisations and allow users to access EHRs when needed. Users are going to be able to download EHRs from physicians and pharmacies and share selected information with family or healthcare providers of their choice. Google started a pilot project in February 2008, in which 1,500 to 10,000 patients' records from the Cleveland Clinic were transferred to Google health (CNN, 2008). These records are accessed by their owners through a user identifier and password. These are the same credentials that users have for all other services provided by Google (i.e. email, and youtube). As this thesis was being written, Google released the Beta version of this service.

Currently, an important disadvantage of using these services is that individuals who use them are not protected by legislation. In the U.S.A. the Health Insurance Portability and Accountability Act (HIPAA) only applies to two types of institutions: healthcare providers (i.e. hospitals, physicians, and health insurance companies), and healthcare clearinghouses. Since customers' EHRs are not covered by HIPAA, there are significant privacy issues. One reason is that EHRs could be used for marketing purposes without the control and support of the legislation.

Nevertheless, legal issues are not the only security and privacy issues while using these types of technologies. Storing huge amounts of EHRs and making them available through the Internet brings vulnerabilities such as, medical identify theft, hijacking, eavesdropping, etc. The use of passwords could also be an issue if users choose easy-to-guess passwords, or the authentication mechanisms are prone to brute force attacks.

3.4 Privacy Laws and Legislation

Healthcare organisations have to protect their customers' healthcare information in order to comply with the security requirements of their customers. In addition, healthcare organisations are governed by laws and legislations in regards to the storing, processing, and transferring of customers' healthcare information. Laws and regulations have to be in place so that users have the support from the government in case breaches in security and privacy occur. Regulations can help to increase healthcare organisations' concern and effort for the protection of their customers' healthcare information.

Organisations have to comply with legal requirements while managing healthcare information in HIS. This research is focused on regulations and legal requirement in the U.S.A. and Australia. In the U.S.A., the HIPAA and breach notification laws are the ones who regulate healthcare organisations. In Australia the Federal Privacy Act and jurisdictional State or Territory privacy and health record laws are the regulators (Liu et al., 2008b).

3.4.1 USA Privacy Laws and Legislation

3.4.1.1 Health Insurance Portability and Accountability Act

HIPAA was instated in 1996 by the USA's Congress with the purpose of supporting healthcare coverage and the establishment of standards for the use of electronic healthcare. The HIPAA has two main sections, "Title I: Health care access, portability, and renewability" and "Title II: Preventing Healthcare Fraud and abuse; administrative simplification; medical liability reform" ("Health Insurance Portability and Accountability Act" [HIPAA], n.d.). Title I of the HIPAA regulates the health insurance plans for workers and their families when they lose their jobs. Title II of the HIPAA defines how to adopt standards for the electronic transaction of certain information.

Title II of HIPAA also known as the "administrative simplification provisions" addresses the security and privacy issues of sensitive healthcare information. The rules specified by the administrative simplification provisions apply to covered entities, which are healthcare clearinghouses (public or private entities that process health information), and healthcare providers (providers of medical or other health services). The administrative simplification provision contains five rules promulgated by the Department of Health and Human Services (HHS), which are as follows (HIPAA, n.d.):

- **Privacy rule.** This rule protects the security and privacy of Protected Health Information (PHI) while maintained by a covered entity. These rules do not specify limits for the transmission of what is called not identified healthcare information, but specify rules for the identification process. The rules specify how the information can be disclosed and used. A covered entity must disclose PHI to facilitate treatment, payment of health operation or in case the individual authorised the disclosure. However, the covered entity must disclose the least amount of information required. Also, covered entities must keep track of the PHI disclosed and must inform owners about any disclosure.

- **The transactions and code sets rule.** This rule sets the standards of transacting medical claims and related business, so that transactions can be more efficient. Covered entities have to adopt standard code sets to be used in all health transactions. All covered entities must use the same transaction coding, thus reducing errors and duplications.
- **The security rule.** Defines the rules to protect all individually identifiable health information (IIHI), which is information that can be related to a specific individual. The covered entities have to ensure the confidentiality, integrity and availability of all IIHI that the covered entity maintains, transmits, creates or receives. This rule also provides support for the privacy rule, in which a covered entity must protect the IIHI in case of any anticipated disclosure that is not allowed by the privacy rule. The rule is scalable, allowing covered entities to select the appropriate technologies according to their operations. Covered entities need to have policies and procedures in place in order to protect IIHI.
- **The unique identifier rule.** This rule defines the standardisation of identification formats in order to reduce errors and confusions. Covered entities must use only the National Provider Identifier (NPI) to identify standard transactions. The NPI is unique to the individual (but not to the organisation) and is comprised by 10 digits without any additional meaning.
- **The enforcement rule.** This rule sets penalties for non-compliance and establishes procedures of investigation. The penalties include:
  - Fines up to $25K for violation in the same calendar year.
  - Fines up to $250K and/or imprisonment up to 10 year for misuse of IIHI.

According to these rules, there are some security mechanisms that have to be implemented by healthcare organisations in order to protect IIHI. HIPAA requires the following mechanisms to be properly implemented in all HIS:

- The final HIPAA security rule requires encryption only when IIHI is sent over a public network, such as the Internet.

- The final HIPAA security rule requires controlled access to the information based on the user’s role within the organisation.
- The final HIPAA security rule requires employing authentication mechanisms such as automatic logoff, passwords, Personal Identification Numbers (PINs) and biometrics in order to identify authorised users and deny access to unauthorised users.

3.4.1.2 State Security Notification Law and AB 1298

Security Breach Notification Laws have been introduced in forty states in the USA as a response to the increasing cases of identity theft. The law was first introduced in California as a bill in the year 2002 and became effective in July 2003. The legislation stipulated that any entity conducting business in California, which stores, manages and transfers personal information of residents of California shall report any security breach to the owners of the information. This legislation known as the “SB 1386” was the first initiative in the USA to enact a breach notification law for financial information. In the subsequent years, this legislation was carried to the other 39 states in the USA forming the basis for the creation of a national notification process under review in Congress. If authorised by the Congress, it will become part of the constitution.

With small differences between the states, generally the laws stipulate that “any business that processes personal information about individual residents of a state must disclose a breach of the security of such information to all such residents affected by the breach” (Hutchins, Caiola, Park, Turner, and Young, 2007).

The law was implemented in most of the states following the Californian model, but there are significant and key differences between the states. These differences could imply a great impact to the appropriate response in case of a security breach.

Generally, the statutes apply to sensitive, unencrypted personal information in which “personal information” is defined as the individual’s first name or first initial, and last name stored with one or more of the following data:

- Social Security Number,
- Driver’s licence number or non-driver’s identification number,
- Any financial account number with or without required password or security access code, or
- Credit card or debit card number with or without required password or security access code.

Personal information is defined differently in various states, for example, in Georgia this information includes a last name and a PIN. Also, in Georgia, the statute states that when there is a breach of even a portion of personal information when encrypted the owner has to be notified. In other states, if the information is encrypted and there is a security breach, they are not required to notify the owners.

The entities that are covered by the statutes also vary from one state to another. Generally, the statutes will cover persons, business and state agencies. However, there are some states that restrict the scope of covered entities by the number of personal information collected (50,000 or 10,000 individuals), or the purpose for which the information is collected (information brokers in Georgia) (Zwillinger, and Sadker, 2005).

A breach can be defined as the unauthorised access or acquisition of computerised data that compromises the confidentiality and integrity of personal information maintained by a person or a business. In general, a notification is sent when computerised data (personal information) was accessed or acquired without authorisation or if there is a "reasonable belief" that the information was accessed or acquired without authorisation. Nevertheless, in some states no notification is sent unless the breach results in substantial harm, but this is not a common approach ("Notice of Security Breach State Laws", 2007).

In October 14, 2007, Governor Schwarzenegger signed the Assembly Bill 1298 which expands the scope of the California data breach notification law to cover medical information and health insurance information. This was made as a response to the recommendations on Medical Identity Theft from the World Privacy Forum. The new law took effect on January 1st, 2008, expanding the definition of personal information to include two data elements defined as follows:

- **Medical Information. "**Any information regarding individual's medical history, medical or physical condition, or medical treatment or diagnosis by a healthcare professional" (Assembly Bill No. 1298, 2007).
- **Health Insurance Information. "**An individual's health insurance policy number or subscriber information number, any unique identifier used by a health insurer to identify the individual or any information in an individual's application and claims history, including any appeals records" (Assembly Bill No. 1298, 2007).

If the AB 1298 spreads through the U.S.A. states in a similar way as the SB 1386, this will represent a major change for all healthcare organisations which manage digital healthcare information. Healthcare organisations may have to revisit their security and privacy policies and standards. Healthcare organisations may have to take some of the following steps:

- Identify and limit the amount of computerised medical information or insurance information that is collected and maintained.
- Ensure that the medical information and insurance information is protected with appropriate security measures.
- Encrypt any data which is considered to be personal information.
- Train human resources, IT personnel, and managers to treat medical information and insurance information as sensitive assets.
- Update the breach detection and response plan.

3.4.2 Australian Privacy Laws and Legislation

3.4.2.1 Federal Privacy Law

In Australia, the principal federal statute which affects private sector business, health service providers, and Australian Capital Territory (ACT) government agencies is the Privacy Act 1988. The Privacy Act 1988 provides protection for the privacy of individuals and similar purposes (Privacy Act 1988, 2007). The Privacy Act 1988 contains eleven Information Privacy Principles (IPP) which apply to the Australian and ACT government agencies, and ten National Privacy Principles (NPP) which apply to the private sector and health service providers.

The Privacy Act 1988 was first created providing the eleven PII which apply to Australian government agencies and the ACT agencies. These principles address the manner in which these agencies can collect, store, and protect personal information. These principles also stipulate the basis in which personal information shall be stored and accessed and the process required to grant access to this information.

From 21 December 2001, the private sector amendments in the Privacy Act 1988 became operative including the ten NPP, found in Schedule 3 of the Act. The NPP apply to private sector organisations with an annual turnover of more than 3 million, and health service providers. For the private sector the NPP determines how authorised organisations are allowed to collect, store and secure personal tax file numbers. Also, the NPP determines how law enforcement agencies can disclose personal information for investigation in case of terrorism.

The NPP monitor the following areas of health service providers:

- The storage, use, disclosure and retention of personal medical information under the Pharmaceutical Medical Scheme and Medicare program.
- The collection, use and disclosure of personal medical information for research, or the compilation or analysis of statistics, relevant to public health or public safety.
- The storage, use, disclosure and retention of personal tax file numbers for those organisations with authorisation to record that information.

The introduction of the NPP has brought forward some issues in regards of the burden of managing healthcare information. Consequently, the Office of the Privacy Commissioner produced guidelines to help private health service providers for the correct operation according to the NPP. In addition, to facilitate medical research without risking personal information, the National Health and Medical Research Council (NHMRC) issued guidelines s.95 and s.95A. These guidelines balance the protection of individuals' healthcare information with the needs for authorised research without the consent of the individuals involved.

3.4.2.2 State and Territory Privacy Acts

The Privacy Act 1988 does not regulate state or territory agencies (except for the ACT). For this reason each state has a privacy act which protects personal information. Some of these acts by state are:

- **New South Wales.** The privacy and Personal Information Protection Act 1998 (PPIP Act) governs how the New South Wales public sector agencies are to store, manage and disclose personal information. The Health Records and Information Privacy Act 2002 (HRIP Act) governs the handling of health information in the public sector and seeks to govern the private sector too.
- **Victoria.** The Victorian Information Privacy Act 2000 (VIP Act) governs the handling of all personal information held by health service providers in the public sector and seeks to govern the private sector. The Health Records Act 2001 provides fair and responsible handling of health information and applies to both public and private sectors.
- **Queensland.** There is no privacy law in Queensland but a privacy scheme and guidelines which applies to the Queensland State Government agencies and other government owned corporations.

- **Western Australia.** There is neither a privacy law nor an administrative privacy regime.
- **South Australia.** There is not a privacy law in South Australia, but there are administrative instructions, in which government agencies have to comply with a set of Information Privacy Principles.
- **Tasmania.** The Personal Information Protection Act 2004 applies to the public and local government sectors and the University of Tasmania.
- **Northern Territory.** The Northern Territory Information Act 2002 governs the protection of personal information, record keeping and archive management of information held in the public sector.

As specified by Liu et al. (2008b) there is not an Australian initiative to create an approach to handle health information legislation like in the HIPAA in the U.S.A. This could cause some issues when creating systems which have to comply with varying regulations between states. Also, security issues could arise when unauthorised disclosure of the information is not properly managed (Liu et al., 2007b).

3.5 Security Requirements and Mechanisms

Extensive use of ICT has brought threats to the security and privacy of patients' healthcare information while stored, processed, or transmitted. Access to patients' information, in the form of EHRs, is the basis for future HIS, but legal and ethical responsibilities exist for the protection of patients' privacy. The sensitivity of the information contained in EHRs has been emphasised by legal obligations of healthcare providers. In the U.S.A., HIPAA mandates that healthcare providers are to protect the integrity and confidentiality of IIHI and also prevents its unauthorised use and disclosure (HIPAA, n.d.).

Healthcare organisations have to mitigate security and privacy threats in order to comply with laws, regulations and ethical standards. Threats in HIS that store, process and transfer EHRs can be listed as follows (Win et al., 2006):

- **Masquerading.** A user or process might pretend to be another entity to gain access to unauthorised files or memory. This also includes medical identity theft, in which a user steals the medical records of another user to obtain medical treatment, prescriptions or medical devices for sale.
- **Infiltration.** This is an attack in which an unauthorised user or process has full access to the resources in a computer system. A buffer overflow attack

could be used to exploit vulnerabilities in an application in order to have access to the resources in a computer system.

- **Unauthorised disclosure of information.** This attack involves unauthorised disclosure of information while is stored, processed or transmitted. Eavesdropping or skimming are common attacks when sensitive information is sent unencrypted through a public network.
- **Unauthorised alteration of information.** This involves unauthorised modification of information while it is stored, processed or transmitted. Tampering is an attack in which an attacker adds information in order to provoke a determined behaviour.
- **Repudiation of actions.** This threat involves the denial that an action ever occurred. For example, health providers could deny the receipt of a payment even if it was done.
- **Unauthorised Denial of Service (DoS).** This involves the denial of services or resources to those who are authorised to access them. A DoS attack could be very serious in real-time healthcare systems.

Unmitigated security and privacy risks can lead not only to legal penalties and prosecution, but also to consequences such as: loss of sensitive information; loss of credibility; and loss of services. Furthermore, once patients' sensitive healthcare information is compromised the damage cannot be undone.

Protecting the security and privacy in HIS is a complex task. Appropriate information security services and mechanisms have to be implemented in order to simplify this complexity. These services and mechanisms have to be focused on the following security objectives:

- Confidentiality and Integrity of data while transmitted.
- Confidentiality and Integrity of data while stored.
- Availability of the information.
- Accountability while working with sensitive data.
- Authentication to accurately verify a claimed identity.
- Non-repudiation.

In the U.S.A., in order to comply with the HIPAA, healthcare organisations have to implement the following mechanisms while protecting sensitive healthcare information (HIPAA, n.d.):

- Encryption mechanisms for the storage and transmission of patients' information.
- Authentication mechanisms to verify the identity to those who access the information.
- Authorisation mechanisms to programmatically determine the rights that are granted to individuals and computer systems.

In addition, any security mechanisms implemented in HIS have to be as transparent as possible for authorised users and be restrictive for unauthorised users.

3.6 Encryption

The use of encryption mechanisms to protect the confidentiality of sensitive information while stored or transmitted is important in large scale systems (Daswani et al., 2007). Encryption can be defined as the process to transform information into unreadable data through the use of cryptographic algorithms. Encryption mechanisms are widely used in commercial organisations to protect the confidentiality of customers' information while stored or transmitted. For example, organisations which accept credit card payments through the Internet use the Hyper Text Transfer Protocol over Secure Socket Layer (HTTPS) protocol. The use of the Secure Socket Layer (SSL) or Transport Layer Security (TLS) protocols allows client and server to negotiate an encrypted communication channel to exchange information. Encryption is also used to protect the information while it is stored in the computer system. By encrypting the data, risks of an unauthorised disclosure are minimised, for example, in case a laptop computer containing sensitive information is lost.

In HIS, encryption is recognized as a proper mechanism to provide confidentiality of healthcare information while stored and transmitted. In the U.S.A., HIPAA mandates that all healthcare information that is stored or transmitted must be encrypted. In a similar way, in order to comply with the AB 1298, healthcare organisations must encrypt any personal information to avoid any unauthorized access or acquisition of the information.

There is a vast diversity of symmetric and asymmetric cryptographic algorithms that can be used to encrypt healthcare information. These algorithms can be summarised as follows (Snyder, and Weaver, 2003):

- **Data Encryption Standard (DES).** This symmetric block cipher processes a 64 bit block of plaintext or ciphertext per iteration. The key size used to encrypt and decrypt the data is 56-bit long. Since the 56-bit key of DES was found to be vulnerable to brute force attacks, DES is not considered to be adequate to protect sensitive information. However, it is still used due to its history as a standard algorithm in the commercial environment.
- **Triple-DES (3DES).** This algorithm was a solution to the 56-bit key of DES without switching to a new algorithm. 3DES makes use of a 168-bit key (three 3DES keys) but is considered to provide an effective 112-bit due to the meet-in-the-middle-attack. 3DES considerably restrains a brute force attack, but is three times slower than DES.
- **Advance Encryption Standard (AES).** This is the “new” Federal Information Processing Standards (FIPS) cryptographic algorithms standard which replaces DES. AES is a symmetric block cipher which processes a block size of 128 bits. AES is capable of using cryptographic keys of 128, 192, and 256 bits. In 2003 the NSA in the U.S.A. announced the acceptance to use AES for encryption of Top Secret information.
- **Rivest-Shamir-Adleman (RSA).** This was the first asymmetric algorithm, which has been widely used for the encryption and signing of digital information. This algorithm is based on the difficulty of factoring large integers and its operations are based on modular arithmetic. The standard key length is 1024-2048 bit long. This is considered to be a secure algorithm as long as it uses a key longer than 1024 bits.
- **Elliptic Curve Cryptography (ECC).** This is an asymmetric cipher which is considered to be the future of asymmetric cryptography. This algorithm is based on the use of elliptic curves over finite fields. This algorithm appears to be more secure with shorter keys than those needed by RSA. Also, this algorithm is supposed to require less process time than other asymmetric ciphers.

Currently, AES is the most widely used cryptographic algorithm due to its strength of all key lengths. In June 2003, the U.S.A. government decided to use AES for the encryption of confidential information. AES has also demonstrated to be a feasible solution for HIS while using a 256-bit key (Weaver et al, 2003).

Asymmetric algorithms have also been used in HIS in order to authenticate users and protect the confidentiality of the information while transmitted. Public Key

Infrastructure (PKI) architectures have been proposed in HIS to provide keys and key management for asymmetric cryptography (Nibbs and Farrelly, 2001). The use of secure protocols such as SSL/TLS is widely used in web based EHR systems (Win et al., 2006). An example of the use of the SSL/TLS protocol is Google Health. Google Health makes extensive use HTTPS to protect the sensitivity of health related information while their customers access their personal health profiles and EHRs.

Encryption is an appropriate solution to protect the confidentiality of sensitive health information while stored or transmitted. However, encryption does not fully satisfy security requirements in HIS. Encryption does not protect the integrity and availability of sensitive healthcare information. If an attacker has access to the encrypted information, the attacker could delete or modify the data without even decrypting the information. In addition, encryption does not restrict the amount of data that the user can access once the information is decrypted. Also, it does not restrict the privileges that users have over the decrypted information. If an authorised user accidentally or intentionally distributes sensitive information, the encryption scheme will fail.

3.7 Access Control

Access control is a fundamental security mechanism in multi-user and resource-sharing computer systems such as HIS (Liu et al, 2007a). The nature of multi-user systems make it very complicated to find a balance between sharing the information and protecting it (Tolone et al, 2005). The purpose of multi-user systems is to make information and resources available to all who need it. On the other hand, the purpose of information security is to protect confidentiality, integrity and availability of information and resources, while restricting access to them. Hence, access control mechanisms are used in multi-user systems to restrict access to the information and resources only to authorised users.

In addition, access control mechanisms have to be used in HIS in order to comply with legal and ethical requirement. HIPAA requires that healthcare providers and clearinghouse have appropriate authentication and authorisation mechanisms to protect patients' healthcare information. Nevertheless, access control has been always an important part of health services, because healthcare professionals have to behave according to strict ethical codes. Ethical codes such as the Hippocratic Oath have to be sworn by every healthcare practitioner. The Hippocratic Oath states that ("Hippocratic Oath", n.d.):

"…Whatever I see or hear in the lives of my patients, whether in connection with my professional practice or not, which ought not to be spoken of outside, I will keep secret, as considering all such things to be private…"

In this statement, healthcare professionals express their responsibility for the information that they hear or see (access permissions) in connection with their professional practice (authorisation). Access control is a native security requirement in healthcare services and consequently a fundamental part in HIS. This becomes even more evident with the increasing use of ICT in HIS. The use of ICT increases the amount of people who can access the information and the speed in which the information can be accessed.

The following sections introduce authentication and authorisation mechanisms that can be used in HIS. Authentication and authorisation mechanisms are a fundamental part while protecting the security and privacy in HIS.

3.7.1 Authentication

In order to restrict access to information and resources only to authorised users, it is important to verify that the claimed identity of a user is legitimate. If authentication mechanisms fail, even if authorisation mechanisms are in place, the security and privacy of the information can still be compromised. Therefore, it is important to accurately verify the identity of a user through the use of various authentication techniques such as: knowledge-based; object-based; ID-based; and location-based. A combination of authentication techniques (two-factor authentication) can be more effective than using a single authentication technique (Daswani et al., 2007). A common example is the use of a PIN number (knowledge-based) and a credit-card (object-based) to withdraw money from an ATM.

In HIS that manage EHRs the most common authentication mechanism is the use of an identifier combined with a password (Win et al., 2006). For example, Google Health clients use their user identifier and password (common to all other Google services) to access their medical profile including EHRs. Smartcards are also used in HIS to store patients' EHRs or for identification purposes. In the UK, HIS that are part of the NHS, make use of smartcards to authenticate staff members (Arnott, 2004). These systems make use of two-factor authentication, associating smartcards with PINs. The PIN is a four-to-eight integer number which is uniquely associated with each smartcard. NHS clinical staff uses the smartcard and their PIN to access sensitive information or restricted resources. Also, HIS make use of biometrics to accurately identify users based on their unique physical characteristics (Weaver et al, 2003). The use of fingerprints, a type of biometrics, is becoming a common authentication mechanism to prevent unauthorised access to sensitive information (Brevetti, 2006). Fingerprints are particularly important in mobile devices such as laptops which can be easily lost or stolen.

The use of PINs and passwords as principal authentication mechanisms provides very weak security protection in computer systems (Win et al., 2006). An attacker could perform a brute force attack in order to break a PIN. PINs constituted by four digits, can be easily broken by an attacker, since the attacker only has to perform 10^4 combinations in order to guess the PIN. In a similar way, if the user selects an easy-to-guess-password, the attacker can commit a dictionary attack and obtain the password.

Authentication mechanisms are an important component while protecting the security and privacy of sensitive healthcare information. However, authentication is not a complete solution since it cannot restrict privileges that users have over the information and resources in the system. For this reason, authorisation is a complementary solution restricting the privileges of authorised users according to the requirements in the security policies.

3.7.2 Authorisation

Cryptography and authentication mechanisms have demonstrated to be key elements in HIS. Nevertheless, authorisation mechanisms must be provided in order to inspect and approve the access rights of users against the level of sensitivity of information and resources (Weaver et al, 2003). HIS that manage EHRs make patients' health information available to a large number of people. Therefore, it is important to ensure that only the right people have access to the right information to protect the patients' privacy. These systems require more than just authentication in order to protect the security and privacy of information. Once the user is properly authenticated, users' operations and accesses have to be restricted according to the organisations' security policies.

HIPAA requires that healthcare organisations provide appropriate authorisation controls to prevent unauthorised accesses to information (HIPAA, n.d.). Inappropriate management of authorisation can lead to legal penalties and prosecution due to the leakage of sensitive information. The lack of adequate authorisation management has been demonstrated in cases such as:

- A former UCLA Medical Centre employee was indicted due to privacy breaches involving at least 61 patients at the UCLA hospital. The employee disclosed information of high-profile patients and sold this information for up to US$4,600 (CNN, 2008).

This is a common scenario in which users exceed their privileges due to lack of proper authorisation management. In order to restrict the privileges that users have over the systems' resources, appropriate access control policies have to be created and enforced through the computer system. Access control policies are enforced by authorisation mechanisms (commonly referred to as access control mechanisms) implemented in the system. These policies have to reflect an access control model in order to define the entities to which the policy applies and the rules that control its behaviour.

The selection of the appropriate model to implement in the organisation depends on the security requirements of the organisation and the security goals they want to achieve. For a healthcare organisation, there are some features to look for while selecting an appropriate access control model, some of these being (Tolone et al, 2005):

- **Generic Implementation.** A model which is able to be configured according to a variety of tasks and organisational models.
- **Transparency.** A model which is transparent for the end-user.
- **Dynamic Implementation.** A model which is able to modify policies at runtime.
- **Reliability.** A model which does not affect the performance or resources.
- **High Level of Specification.** A model which defines access rights at a high level of specification (understandable for the end-user).
- **Greater scalability.** A model which is able to support an increasing workload.

RBAC is the dominant model for advanced access control in HIS. RBAC supports security principles such as the principle of least privilege, specification of competency to perform specific tasks and enforcement of conflict of interest rules. In addition, RBAC simplifies the complexity and total costs of security administration in large scale systems. Healthcare providers can use RBAC to conform to the requirements of regulations such as HIPAA. HIPAA requires the use of authorisation mechanisms that restrict access based on the users' role within the organisation. For example, a nurse can only access the role 'nurse', which can be limited only to add entries to the patient's EHR, instead of having access to the whole record. Consequently, patients' healthcare information can be accessed without being completely disclosed.

Nevertheless, RBAC is considered to be insufficient to fully satisfy access control requirements in healthcare environments (Hu and Weaver, 2004). RBAC do not take into consideration the context in the activation, and deactivation of access permissions. For example, if a doctor is absent, there is no way in which another doctor can take his privileges over his/her resources in an emergency. There are some initiatives to create a better access control model for HIS, such as the one proposed by Schwartman, D (2004), Hu and Weaver (2004) or Bacon et al. (2003). However, there is not yet a model which can be considered the most suitable for HIS.

3.8 Discretionary Access Control in HIS

It is important to provide appropriate access control mechanisms at the OS layer due to the level of granularity that can be applied while restricting access to resources. The OS manages hardware components such as the disk drive, the keyboard, network cards, the monitor and the printer. The OS is the one that mediates every access request from applications running in the systems to system resources. Therefore, access control mechanisms at the OS layer can prevent any unauthorised access to the systems' resources. The granularity at the OS layer can restrict access to system resources such as: data and executable program files, input/output devices, network components, software processes, etc.

The majority of current information systems, including HIS, are constructed based on OSs that use DAC mechanisms to restrict access to the resources (Liu et al., 2008a). OSs that implement DAC mechanisms are common off-the-shelf products. This includes commercial products such as MS Windows and open source products such as Linux. These systems are characterised by the fact that access to resources is at the discretion of the users. This means that users are authorised to grant and revoke access permissions over the resources that they create, use, or own, without regards to security policies. Also, commonly these OSs have only two levels of privileges, which are: the system administrator and the ordinary user level. Users with system administrator privileges have full control over the entire system, including unlimited access to resources. On the other hand, those at the ordinary user level have full control only over the data and program files that they create, use, or own.

Nevertheless, OS that enforce DAC are ineffective while protecting the resources that they manage (Loscocco et al, 1998). The following section is dedicated to demonstrate the security issues while using OS that implement DAC mechanisms.

3.8.1 Discretionary Access Control Issues

The first issue with DAC is that usually those who are authorised to access the information do not own the information (Reid et al., 2003). In HIS, patients are the ones who own their healthcare records. However, doctors and nurses have to access patients' healthcare records in order to add diagnostics and/or observations. In a DAC system this will imply that those authorised to access patients' healthcare records can grant access permissions over these resources to others at their discretion. Because of the security and privacy requirements of healthcare organisations, those who create, use or own the information should not have the discretion to grant or revoke access permissions over the resources to others.

Access permissions over the resources have to be granted according to the access control model specified in the security policy. In this way, if a physician creates healthcare records as files, the physician should not be able to pass permission over these files to other physicians. This is not possible in discretionary access control models such as DAC, but is possible with non-discretionary access control models such as RBAC.

Access control mechanisms at the OS layer have to enforce the security policy. In this way, the granularity while accessing system resources relies on the security policy and not the discretion of the owner of the resources. This being said, even users with system administrator privileges should be restricted according to the security policy.

The second issue in DAC systems is the enforcement of least privilege principle, which is absent in these type of systems (Henricksen et al., 2007). The existence of only two levels of privileges in DAC systems does not satisfy the least privilege principle. The least privilege principle states that users must be restricted to the least number of privileges required to achieve their activities. In OSs which implement DAC, the system administrator is not restricted to the least number of privileges; on the contrary, he/she has full control over all the resources in the system. If an attacker gains system administrator privileges, the attacker would be able to compromise the entire system. In HIS, it is desirable to restrict privileges that users have over the resources according to their job functions (Bennett, Rigby, and Budgen, 2006). For example, it would be desirable that physicians are the only ones allowed to modify healthcare records, while nurses should only be able to append the patients' status in the records. Restricting full access to the resources can prevent unauthorised disclosure of information. Inappropriate enforcement of the least privilege principle has been demonstrated in cases such as:

- Kaiser Foundation Health Plan Inc. received a fine of US$200,000 when a disgruntled employee posted confidential information about patients on a Weblog. The information posted included medical record numbers, patient names, and in some cases routine lab tests (Ostrov, 2005).
- A former UCLA Medical Centre employee was indicted due to privacy breaches involving at least 61 patients at the UCLA hospital. The employee disclosed information of high-profile patients such as Britney Spears (CNN, 2008).

In order to prevent similar security breaches, appropriate enforcement of the least privilege principle needs to be specified in the security policies. OSs must provide access to resources that they manage according to the least privilege principle. This means that, the level of granularity is extended so that users have access over system resources (files, directories, processes, etc.) with the least number of privileges. For example, physicians may be able to create files in directory A, while nurses can only append the data in the files of directory A. In a similar manner, physicians may be able to read and write in a specific network socket, while nurses are only authorised to write in the same network socket.

The third issue in DAC systems is the lack of domain separation. Domain separation consists of the creation of sandboxes which limit the operations that users are allowed to perform with resources inside the boundaries of the sandbox (Henricksen et al, 2007). If the users' account is compromised, the sandbox restricts the damage in such a way that processes and resources outside the sandbox boundaries are not affected.

In any computer system, the user is not the one who has direct contact with the system resources. An application, which operates on behalf of the user has the direct contact. Consequently, the application acquires the privileges of the user and is able to access all the resources authorised to the user. Therefore, if an attacker is able to compromise the application, the attacker will be able to acquire the same privileges as the user. In an OS that implements DAC, this would mean that the application can interfere with other applications which are run by the user or access resources not required by the application. In addition, because DAC only has two levels of privileges, an attacker could escalate privileges to grant system administer privileges. That is, once an application is compromised, the attacker could interfere with other applications until the attacker gets system administrator privileges. If this is the case, the attacker would be able to compromise the whole system. Attackers exploiting vulnerabilities in HIS applications have been demonstrated in cases such as:

- In the Colorado University in the USA, names, SSN, addresses, and dates of birth of 42,000 individuals plus 2,000 lab test results were stolen by a hacker who broke into the healthcare servers ("Summit Daily News", 2005).
- In the Duke University Medical Centre in the USA 5,000 users' passwords and nearly 9,000 fragments of SSN, which belonged to medical school alumni, medical centre staff, faculty and trainees, were stolen by a hacker who broke into the computer system (Dixon, 2006).

The use of sandboxes can restrict this type of attacks, by restricting the compromised application to the resources inside the boundaries of the sandbox. This would also make it impossible for an attacker to escalate privileges by interfering with processes in the system. OS that implement DAC do not provide this type of protection.

DAC is not enough to satisfy security requirements in HIS. Healthcare organisations have to comply with strict security requirements, which are not satisfied by off-the-shelf products such as MS Windows or traditional Linux. In HIS, MAC-based OSs are a preferred solution to satisfy security and privacy requirements for information resources (Liu et al., 2008a). In OSs which enforce MAC, access to resources is not at the discretion of the individual user, but of the security policies enforced by the OS. MAC mechanisms can provide protection against intentional or unintentional attempts to destroy or damage the system. Attacks from external attackers can be restricted by the use of sandboxes, while attacks from insiders can be restricted by the enforcement of the least privilege principle. OSs which implement mechanisms that allow the enforcement of a RBAC policy are able to appropriately satisfy the least privilege principle. Currently, there are several OS which can provide MAC mechanisms such as Red Hat Enterprise Linux V5 (RHEL5), Fedora Core 8, or Solaris 10. From these OSs, Fedora Core 8 has demonstrated to be an appropriate research solution. This is because Fedora Core 8 is freely available with SELinux enabled by default. SELinux is a variation of Linux that implements a flexible and fine grained MAC in the Linux kernel. This research is focused on the viability of using SELinux to introduce MAC at the OS layer in HIS.

In the past, a similar research was made by Henricksen et al. (2007). They proposed a holistic approach to protect health-care records, based on a three layered architecture. In this approach, SELinux is introduced as a technology which can provide "high-grained flexibility and architectural modularity for securing application data" (Henricksen et al., 2007). However, several identified issues while using SELinux made it not totally appropriate for HIS. This research used early releases of SELinux. Consequently, the issues mentioned by Henricksen et al. (2007), such as inflexible modification of policies and troublesome policy management, can be solved with tools and technologies that are now available with newer releases of SELinux. SELinux is a technology that has changed dramatically since its release in 2003 due to the increasing interest from researches and the open source community. Tools and technologies are constantly developed to address complexities in the use of SELinux and to make it more feasible for commercial purposes. This being said, there have been tools and technologies released in the last years which were not available to Henricksen et al. (2007) at the time of their research. For example, Henricksen et al. (2007) stated that SELinux is not able to "gracefully change policies when circumstances change". However, as demonstrated by this research, this issue can be addressed by using conditional policies in current versions of SELinux.

A similar proposition is made by Liu et al. (2008a). Liu et al. (2008a) proposed the use of SELinux as part of the security mechanisms used in the Open Trusted Health Informatics Structure (OTHIS) architecture. They already made other distinctions of the importance of SELinux and MAC mechanisms at the OS layer for HIS in previous publications (Liu et al., 2008, b; Liu et al., 2007a; Liu et al., 2007b). However, these findings do not clearly explain how SELinux could be implemented and managed to satisfy security requirements in HIS.

For this reason, this thesis proposes a modern framework based on SELinux Profiles to aid in the implementation and management of SELinux in HIS. This thesis updates and complements research made in the past which proposes SELinux as a viable solution to provide MAC at the OS layer in HIS.

3.9 Conclusion

This chapter described the security and privacy requirements in HIS along with current approaches to satisfying these requirements. In order to take full advantage of the benefits of ICT in HIS, patients' information has to be available at the point of care. However, making the information available increases the risks of an unauthorised disclosure or modification of information. There are three main security mechanisms that need to be implemented in order to satisfy security requirements in HIS: encryption, authentication and authorisation. However, authorisation mechanisms are the key to prevent unauthorised disclosure and modification.

To appropriately protect resources in a system from unauthorised accesses, access control mechanisms have to be implemented at the OS layer. This can restrict damages if an application is compromised. Currently, most systems are constructed using OS which implement DAC mechanisms (i.e. Windows). DAC is not enough to satisfy security requirements in HIS. This is due to several issues, such as: lack of control while users grant access permissions; lack of domain separation; and lack of enforcement of the least privileges principle. 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 has been proposed in the past to provide MAC at the OS layer to protect healthcare information. However, previous research does not clearly define how the latest releases of SELinux can be implemented and managed to satisfy security and privacy requirements in HIS. In order to update and complement earlier research, this thesis provides a modern framework to implement and manage SELinux in HIS.

Chapter 4

SELinux to Enforce Mandatory Access Control in Health Information Systems

Contents:

4 SELinux to enforce MAC in HIS

4.1 Introduction

Access to applications and resources has to be controlled in a granular and flexible way in order to protect the security and privacy of sensitive healthcare information. 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 organisations 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 the previous chapter security mechanisms were introduced such as encryption, authentication and authorisation mechanisms. Nevertheless, 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 organisation. The majority of information systems, including HIS, are constructed based on OSs that implement DAC mechanisms. However, there are security issues in regard of the use of DAC at the OS layer. 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. In addition, this type of OSs can provide enforcement of the least privilege principle and domain separation which are important while protecting the security and privacy in HIS.

In this chapter SELinux is introduced as an alternative to off-the-shelf OSs which implement DAC. 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 (Wright, Cowan, Morris, Smalley, and Kroah-Hartman, 2002). Through the use of the LSM framework, SELinux was introduced reducing significantly the impact on the

Linux Kernel. This provided SELinux with a reasonable flexibility while enforcing different policies in the kernel and server level.

4.2.1 Linux Security Module Framework

Cowan (2001) proposed to start the development of the LSM framework as a way to address the need to support a variety of security enhancements in the Linux Kernel. The main objective was to create a framework which can: support different security models; not impact in the Linux kernel; and supports the existing POSIX capabilities.

The result of this effort was 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. As shown in Figure 1, a user request to read, write or process any of the resources in the system has to pass a set of steps before the permission is granted. In a standard Linux system the request has to pass the traditional Linux Kernel logic and error handling followed by the standard Linux DAC checks. With the LSM hooks, access to the resources is granted after the LSM hook is called. These hooks are set just after the Linux DAC checks and before accessing the requested object.

Figure 1. LSM Hooks Architecture (Wright et al., 2001)

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 early stages of SELinux, the NSA based the design of SELinux in microkernel projects. The Flask microkernel-based OS was a prototype proposed in 1999 to demonstrate the feasibility of policy flexibility using the Flask Security Architecture (Spencer, 1999). SELinux adopted the Flask security architecture which allows security policy flexibility along with features such as: controlling the propagation of access permissions according to the security policy; supporting dynamic policies through revocation of previously granted access rights; and providing fine-grained access controls.

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 customisation. 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, labelling and poly instantiation decisions. The security server makes the decisions based on the security policy loaded through the policy management interface. These decisions are made based on a pair of entities, usually a subject and object. The security server also provides the labels to be assigned to an object in the system.

- **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 labelled with a set of attributes known as the security context. The Flask architecture provides two data types for object labelling which are (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 context has to be treated as an opaque data type so that it can be handled by the object managers without compromising the policy flexibility of the object manager.

- **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 that represents the security context in 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 computerised 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. That is, the AVC can coordinate requests from the object managers for policy decisions and requests from the security server for policy changes. When client requests access permissions over an object, the object manager sends a query to the AVC containing the triplet of source SID, target SID and requested access permissions to the AVC. If the entry exists, the AVC returns the access decision. If the entry does not exist, the AVC sends a query to the security server which checks the policy logic and returns the access ruling.

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 2, 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.

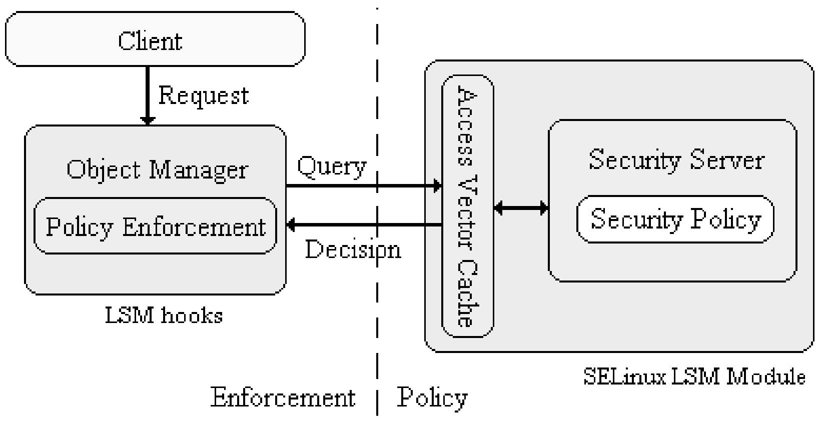


Figure 2. SELinux LSM Module and the Flask Architecture

4.2.3 User Space Object Managers

The Flask security architecture brings one of the most important qualities of SELinux which is the use of user-space object managers. Because the Flask security architecture specifies a clear separation between policy enforcement and security policy decision making, the object managers could be created for user-space servers. An object manager which executes user-space is called a user-space object manager. Object managers could be specified for DB servers or web servers.

In SELinux which is implemented in the Linux kernel through the LSM framework, security objects are represented by the LSM hooks. In SELinux there are different LSM hooks for kernel-level objects such as the filesystem, process management or interprocess communication (IPC). Each of these LSM hooks represents a kernel-space object manager which controls the policy enforcement over the objects that they control. The policy decision making is achieved by the kernel-space security server in which the set of rules comprising the security policy are implemented.

As shown in Figure 3, in the case of the user-space object managers its functionality is similar to those of the kernel-space object managers, but with two main differences. First, the user-space object managers are not represented by the LSM hooks. Second, the user-space object managers do not use the kernel-space AVC.

Libselinux is the library which is used for the support of user-space servers. This library provides the user-space AVC functionality which interprets the SID at the user-space level. The LSM hooks are replaced by interfaces which are internal to the user-space object managers to interact with the user-space AVC inside the libselinux libraries.

The user-space object mangers make use of the kernel-space security server to determine the access decisions for requests from the clients. Object classes have to be specified in the security policy within the kernel-space security server in order to represent internal abstractions and resources embraced by the user-space object manager. Object classes have to be created in such a way that they do not overlap other object classes from other user-space object managers. That is, if a user-space object manager requires an object class “table” for a DB, the object class has to be created so that it does not overlap another object “table” used in another user-space object manager. If this happens, the kernel-space server manager provides no way to manage this conflict. Consequently, permissions could be granted to unauthorised processes.

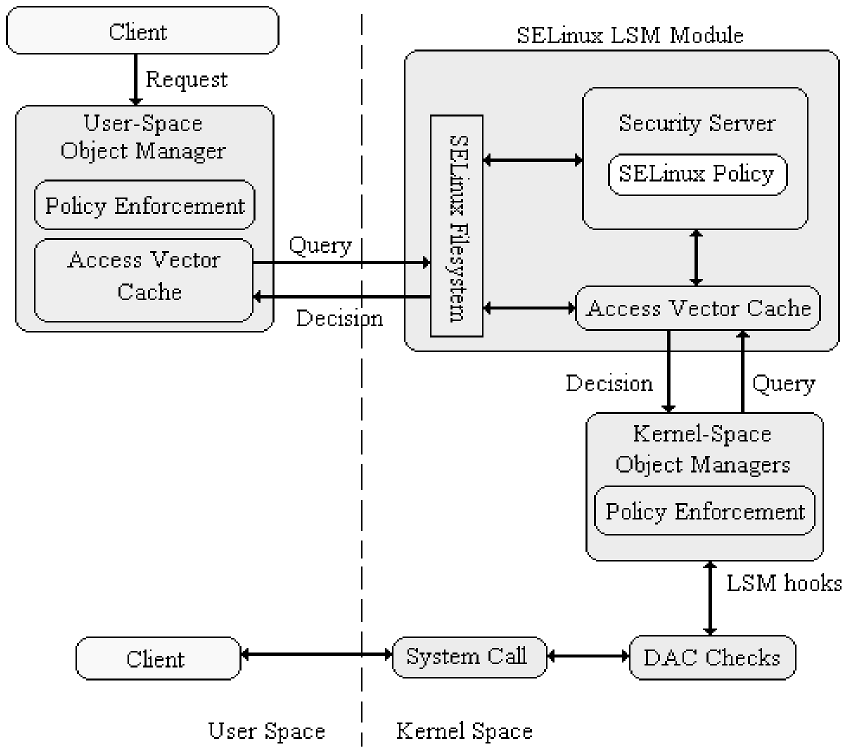


Figure 3. User-Space Object Managers

4.2.4 Policy Management Server

Tresys Technologies with support from the SELinux community has been developing a project which addresses some issues while implementing user-space object managers (“*SELinux Policy Server*”, n.d.). This project also provides granular protection for

SELinux policy management. Basically the policy management server has two functions which are:

- **Policy Management.** This refers to the protection of the SELinux policy through the encapsulation of the policy, fine-grained access control while updating the SELinux policy and a better policy management infrastructure.
- **Access decision support for user-space object managers**. This is achieved through the implementation of a copy of the kernel-space security server into a user-space daemon.

These functions are basically achieved through the use of two servers, the policy server and the user-space security server.

The goal of the policy server is to encapsulate every access to the system policy. That is, every request for an access decision has to be mediated along with the updates to the system's policy. In the current implementation of SELinux any domain with write permission to the binary policy (monolithic binary file that is loaded into the kernel) is allowed to add, modify, or delete portions or the whole system policy. The policy server provides a more granular access control over the system's policy. This is achieved through the delegation of different policy administration tasks to different domains. For example, a DB domain could be allowed to modify TE rules corresponding to its object classes but not the ones of other user-space servers or the kernel. In the same way, a domain could only modify RBAC rules, but not TE rules.

The user-space security server is responsible for access decisions made over user-space object classes keeping separated the kernel-space object classes and the user-space object classes. In this way the kernel-space security server does not need to be aware of the existence of the user-space object managers and their object classes, reducing the impact on performance and memory storage. The user-space object managers query the user-space security server for access decision and cache them in their own AVC mitigating the access decision processing overhead.

The current release of the Policy Management Sever by Tresys Technologies is still considered to be experimental and is not recommended for production environments. This release is only for testing purposes, so that it can be used to demonstrate the management and enforcement capabilities of the server. However, in the future this could be the default architecture in future releases of SELinux.

4.3 SELinux Labelling

As specified by the Flask architecture, every object and subject in the system is associated with a security context. The security context contains security attributes that are only understandable by the security server. In SELinux the security context is comprised by three colon separated ASCII strings as security attributes which are the values for identity, role and type. The security contexts are assigned to objects and subjects in the system according to the rules in the system's policy. In an OS with SELinux enabled, a security context for a file in the home directory looks as follows:

user_t:object_r:unconfined_t

In this example the SELinux user identifier (explained in more detail in Section 4.5) is *user_t*, the role of the object is *object_r* (which in SELinux is always the same for objects) and the type is *unconfined_t*. These three attributes are used while taking access control decisions for domains accessing objects in the system.

An important change brought by SELinux while labelling files in Linux systems is the use of extended attributes (xattrs) as an extension to normal inode-based attributes. In traditional Linux systems an inode is used as the unique identifiers for a file. Inodes contain critical metadata for the file such as UNIX ownership data and access control information. SELinux makes use of the xattrs to add security features into the Linux filesystem. The xattrs are name-value pairs associated permanently with files and directories. SELinux makes use of the extended security attribute namespace *security.selinux* to store the security context labels associated with specific files. For this reason SELinux can only be used by filesystem that support xattrs such as ext3, ext2, and XFS. The use of xattrs allows the preservation of security contexts even during backup and restoration processes.

4.4 SELinux Access Control Mechanisms

SELinux implements a MAC called TE along with a form of RBAC and optional MLS and Multi Level Category (MLC). The TE implemented in SELinux allows no access by default. For this reason, every access has to be specified with a TE rule thus providing a fined-grained access control to protect processes and objects. SELinux implements a form of RBAC built over TE which helps to simplify the complexity of user management in large scale systems. Optional MLS and MLC are provided but these models are out of the scope of this research for being considered to be unsuitable for HIS.

4.4.1 Type Enforcement

SELinux implements a fine-grained MAC model called TE. In this model every object and subject in the system is respectively labelled with a type or a domain. In the TE model, every subject (process) is labelled with a domain and every object (files, directories, etc.) is labelled with a type. However, SELinux manage domains in a different way than traditional TE. As in traditional TE, in SELinux a domain is just a type which can be associated with a process. However, internally domains are treated in the same way as types. The term domain is preserved in the literature to avoid confusion between types for subjects and types for objects. In SELinux, the type is assigned to subjects and objects as a single security attribute in the security context. This type along with the user identifier and the role are the security attributes used by the security server to take access control decisions based on the system's policy.

In contrast to traditional TE which uses two matrixes for access decision making, TE in SELinux only makes use of one. In the traditional TE, the DDT is used to determine the types to which the domains are allowed access, and the DIT is used to specify interactions between domains. Since SELinux does not make any internal distinction between domains and types a single matrix is used to specify interactions between different types. This matrix is contained in the Security Server and is constructed from all the rules contained in the SELinux policy. The use of only one matrix for access decisions help to improve the performance in which access decisions are taken.

A characteristic of TE in SELinux is the use of object classes which are a consequence of using the Flask architecture. Object classes are used to group objects of the same category such as files, directories, sockets, etc. The use of object classes helps to increase the granularity of the model. Objects with the same type can be treated differently if they have different object classes. For example, the policy will be able to recognise block files and character files created by the same domain.

One of the most important characteristics of TE in SELinux is that no access is allowed by default. Every access in SELinux has to be explicitly granted using TE rules. TE rules can be grouped in two basic categories (Mayer, MacMillan, and Caplan, 2007): access vector and transition rules.

Access Vector Rules (AVRs) are those used to authorise access permissions for subjects over objects. There are different types of AVRs including allow, auditallow, dontaudit, and neverallow rules. Any of these AVRs is comprised by a source and target types, the object class to be accessed and the access permissions. If no matching rule is found while taking an access decision, access is denied by default. Every denied access is audited except when the dontaduit rule is specified and no authorised permission is audited except when the auditallow rule is specified. The most common AVR is the allow rule which is declared as follows:

allow subject_t object_t: file { read write getattr };

In this example the allow rule is authorising subjects with the domain *subject_t* to have *read*, *write* and *getattr* permissions over files with type *object_t*. Note that if the object class changes to *dir*, then all subjects with the type *subject_t* can *read*, *write* and *getattr* directories with the type *object_t*.

Transition rules are those which specify the default type to be assigned during object creation or domain transition. A domain transition is a change of domain when a process does an *exec* command. During a domain transition the default type is assigned based on the current domain of the process and type of the program executed. During an object creation the default type is assigned based on the domain of the creating process, the type of a related object and the object class. If no matching rule is found, the created object acquires the type of the related object and for a domain transition the type of the process remains unchanged. An example of a transition rule during object creation is as follows:

type_transition subject_t object_t : file new_object_t

In this rule when a subject with type *subject_t* creates a file in a directory with type *object_t* by default, the new created file will have the type *new_object_t*. In cases where the object class is changed to *dir*, subjects with the type *subject_t*, when creating a directory inside directories with type *object_t* by default, the new directory will be created with the type *new_object_t*.

The type transition rules have to be accompanied by the appropriate AVRs to allow the transitions to take place. For example, let us assume that there is a shell process with domain *hc_doc_t* which needs to transit to the domain *hc_diag_sys_t* belonging to an application process (this is an example of a user executing an application from the command line in a Linux shell). In addition, the executable file of the application is labelled with a type *hc_diag_sys_exec_t*. In order to allow the domain *hc_doc_t* to transit into the domain *hc_diag_sys_t* while executing a file with type *hc_diag_sys_exec_t*, the following TE rules are required:

1. allow hc_doc_t hc_diag_sys_exec_t:file { getattr read execute };

2. type_transition hc_doc_t hc_diag_sys_exec_t:process hc_doc_diag_t;

3. allow hc_doc_diag_t hc_diag_sys_exec_t:file entrypoint;

3. allow hc_doc_t hc_doc_diag_t:process transition;

The first rule authorizes the shell process domain *hc_doc_t* to execute files with the type *hc_diag_sys_exec_t*. The second rule defines the default behaviour for domain transition; in this case to transit into the application process domain *hc_doc_diag_t* when files with type *hc_diag_sys_exec_t* are executed by processes with domain *hc_doc_t*. The third rule provides *entrypoint* access to the *hc_diag_sys_exec_t* domain, that is, this rule allows the executable file to run in the *hc_doc_diag_t* domain. The last rule allows the shell process domain *hc_doc_t* to transit into the application process domain *hc_dog_diag_t*.

4.4.2 Role-Based Access Control

RBAC in SELinux is built upon TE. In contrast to traditional RBAC, roles are not used to group permissions to which the users are authorised. In SELinux users are associated with certain roles which group the set of domains to which users are authorised to access. It is important to notice that roles are used to group a set of domains but not object types. That is, users are restricted by their authorised roles to processes, not to objects in the system. The security context of a process will contain the role of the user that is executing the process. For objects, the role attribute in the security context is not used and therefore is set to a generic role called *object_r*.

In SELinux users are assigned to roles using the *user statement* which specifies the roles that are allowed to coexist with the SELinux user identifier in a security context. For example, the statement *user user_u roles { role_r }* will create a SELinux user identifier *user_u* which is authorised to access the role *role_r*. Therefore the security context *user_u : role_r : any_type_t* is possible, any other role different than *role_r* is not authorized to coexist in a security context with the user identifier *user_u*.

In a similar manner domains are assigned to the role using a role statement and only the domains specified in this statement are allowed to coexist in a process security context. For example, the statement *role role_r types user_t* authorises the type *user_t* to coexist in a security context with the role *role_r*. Any other type is not authorised.

An important feature of RBAC in SELinux is the existence of role dominance statement. This statement allows declaring a role in terms of other roles creating a hierarchical relationship among them. In this hierarchical relationship the role with the higher hierarchy or dominant role inherits the type associations of the roles it dominates.

The use of RBAC along with the TE features provide a model which reduces the burden while managing users in large scale systems with fine-grained protection over subjects and objects. Another benefit of RBAC in SELinux is the role hierarchies which have the potential to simplify the policy configuration.

4.5 User Identifiers

Users in SELinux are managed as orthogonal UID to those provided by traditional Linux systems. In traditional Linux systems, users log in using the string representation of the UID and a password. The UID is an integer that is used by the Linux system to determine access permissions over the resources in the system. Users can change the UID at any time they want using the *su* command or any set UID call. This possibility of changing the user identifier complicates the ability to track activities of logged in users for auditing purposes and authentication.

SELinux UIDs are mapped to traditional Linux UIDs and maintained in the user security attribute in the security context. SELinux UIDs are totally independent from Linux UIDs so that the SELinux MAC remains orthogonal to the Linux DAC. If a user changes its UID through the use of the *su* command, the SELinux UID remains constant even if the Linux UID changes. This is a highly desirable characteristic in order to achieve user accountability and prevent other security violations. This separation of UIDs also provides flexibility with the definition of the SELinux user attribute. SELinux could change its user identifier attributes without affecting its compatibility with Linux UID semantics.

Another characteristic while allowing separation of identifiers is that a SELinux UID could be mapped to more than one Linux UID. This allows grouping users with the same access requirements and mapping them to the same SELinux UID. Doing this reduces the difficulty while managing users in large scale systems. Currently for example, users that are not specified in the Target Policy in Fedora Core 8 are assigned to the same unconfined user which works as an unprivileged user in traditional Linux systems.

4.6 Conditional Policies

One of the most important features in SELinux is the support for conditional policies (Mayer et al., 2007). Conditional policies allow runtime modifications of the system's policy without having to reload the whole SELinux policy again into the kernel. This behaviour consists in sets of policy rules within the SELinux policy which are enabled or disabled using boolean variables and conditional statements. In a healthcare environment, if an emergency occurs, sections of the SELinux policy could be enabled allowing access to sensitive information which in normal situations would be restricted. In a different scenario, in which SELinux is used in a laptop, maybe the laptop requires enabling pieces of the SELinux policy to restrict access to resources available only for mobile workforces.

Boolean variables are defined using the *bool* statement specifying the name of the boolean and its initial value. As an example, the statement *bool hc_emergency false* defines a boolean named *hc_emergency* with an initial value of false.

Conditional statements make use of conditional expressions and boolean variables to evaluate the use to AVRs. These are similar to conditional statements in higher level programming languages such as the C language. In such statements if the conditional expression in the statement is true, then the list of rules within the conditional statement are enabled. On the other hand, if the conditional expression in the statement is false, the rules are disabled.

Once the policy is loaded into the Linux Kernel the values of the boolean variables can be modified at runtime by writing to files existing in the */selinux/booleans* directory. Files inside this directory correspond to each boolean in the SELinux policy. Also, there are command line tools that can be used to modify boolean values such as the *setbool* command. An example of how to modify these values is as follows:

```
echo 1 > /selinux/booleans/hc_emergency

echo 1 > /selinx/commit_pending_bools
```

The first line sets the boolean variable *hc_emergency* to true, while the second line is used to commit any change made in the boolean variables. That is, if the second line is never executed, then the value of the boolean would never change. Note that in order to set the boolean variable *hc_emergency* to false then 0 would have to be used instead of 1 in the first example line.

4.7 SELinux Security Policy

The main distinction of SELinux, from other OSs which implement MAC mechanisms, is that MAC rules are not hard-coded (Mayer et al., 2007). That is, the policy can be configured according to different requirements depending on the security policies of different types of organisations. This characteristic makes SELinux a very flexible solution while implementing MAC to protect resources in information systems. In addition, through the use of SELinux TE, in which no access is allowed by default, organisations can define fine-grained access permissions to sensitive information.

The SELinux policy is a binary file that contains all the access control rules to be enforced by the Linux kernel through the use of the Security Server and Object Managers. The SELinux policy is loaded during the boot process into the kernel for posterior access control decision making. The binary file is created through the use of the *checkpolicy* command tool and a set of files which specify the TE rules, RBAC rules and label assignment specifications. These files are compiled in a similar manner as with higher-level compilers, using the *checkpolicy* command, and integrated in a single binary file. The binary file could be found in the directory corresponding to the name of the enforced policy within the *etc* directory and with the name *policy.[version]*.

In the past, SELinux policies were managed as Monolithic Policies. A Monolithic Policy is a single binary policy file created by the *checkpolicy* command and loaded into the Linux kernel. The binary file is the result of the compilation of a large single source file which was the result of the concatenation of different source modules through the use of scripts, *m4 macros* and *Makefiles*. The main issue while loading a single binary policy into the kernel is that any change required in the policy (even the most insignificant change) would require recreating the whole process again. That is, every change would require the concatenation of all the source modules to create the source file, the compilation of the source file into a single binary file and finally loading the binary file into the kernel. All this process is tiresome, prone to errors and complicates the development and maintenance of SELinux policies.

4.7.1 Loadable Policy Modules

Fedora Core 5 and RHEL5 introduced the use of loadable policy modules to simplify the development and maintenance of SELinux policies. With loadable policy modules, SELinux policies are created as a group of modules. Modules are created independently one from the other as non-base modules and the main binary file is created as a base module. The base module is the one that is first loaded into the kernel and can interact with non-based modules. The base module (binary file) can be created with only specific rules, for example, only those rules required by the proper operation of the Linux OS. All other rules required by the organisation can be loaded into the kernel as non-based modules that interact with the base module.

The base module and the non-base modules can be created using the *Checkmodule* command line tool existing in current versions of Fedora Core and Red Hat Enterprise Linux (RHEL). This command will compile a security policy module (which is a Reference Policy Module) into a binary representation. Another novelty in these versions of Fedora Core and RHEL is that based and non-based modules are loaded into the kernel as Policy Packages. Policy Packages are created using the *semodule_package* command line tool from the binary representation of the security module and other data such as file contexts. These packages are loaded into the kernel using the *semodule* command line tool.

The main goal of the loadable policy modules is to help system administrators to deploy, modify and update SELinux policies in a secure and convenient manner. In the past with the use of monolithic policies when the policy was modified it was required to re-compile the whole policy and reload it into the kernel. With loadable policy modules a change will require to modify only the related module and reload it into the kernel without having to reload the whole policy. That means that the modules once loaded are enforced immediately by the Linux kernel. Therefore, there is no need to interrupt services or disrupt users from their activities.

4.7.2 Security Policy Issues and Tools

Due to the fine-grained access control introduced by SELinux TE, thousand of rules have to be specified reflecting every access required by the computer system to operate properly. The SELinux security policy included in a Linux distribution contains approximately 54,000 rule statements, 100 distinct permissions, and 700 type definitions (Herzog, and Guttman, 2002). This fact makes the task of a system administrator very complicated while analysing and managing the SELinux policy. A policy with so many lines can be prone to errors in which permissions could be granted to unauthorised applications without the System Administrator realising it. The difficulty of the SELinux policies has been considered one of the major factors preventing the adoption of SELinux (Kuliniewicz, 2006).

In recent years there has been a great interest to minimise the burden of managing SELinux policies by the SELinux community and other institutions ("*SELinux Symposium*", 2008). Currently there are many tools being developed, with more to come, but SELinux is still a young technology and so are the tools to work with it. The development of tools is a trial and error task with an increasing interest from private organisation and the open source community.

Currently there is a list of tools important for the management of SELinux policies. These tools help to simplify the creation, analysis, testing and maintenance of SELinux policies. These tools are:

- **SLIDE.** Tresys Technology created a plug-in called SLIDE which works with the Eclipse Software Development Kit (SDK) for creating, editing and testing Reference policies for SELinux.

- **PolGen**. This is a tool that can be used not only by System Administrators but also by application authors for automated SELinux policy generation. PolGen uses a modified *strace* tool called *stracesc* to identify dynamic behaviour of a targeted application. The modified *strace* displays the security context of the executing process and list all the resources accessed. PolGen has a user interface that allows the user to fill gaps in the assumptions made by the tool while developing the policy.
- **Apol.** It is an analysis tool part of the *SETools* toolkit which allows a complete analysis of a SELinux Policy. Apol makes use of the binary file loaded into the kernel to identify the existing users, roles, types, AVR, etc. and their interactions between them. Apol is also able to analyse boolean variable and change their values in order to identify authorised accesses once the boolean is enabled. Also, Apol allows committing information flow analysis between domain and types in the SELinux policy.
- **Setrubleshoot**. This is a tool that exposes AVC denials in real time and interacts with the user presenting the current denial and presenting possible solutions
- **Dynamic Policy Enforcement Agent (DPEA).** These agents can be used along with the conditional policies to automatically enable or disable pieces of the policy according to specific changes in the system's environment. These changes can be detected, for example, by analysing the logs generated by an Intrusion Detection System (IDS).

The use of these tools simplifies the development and management of SELinux Policies but the race for the development of tools for SELinux has just started. In the future there is a high expectancy of better tools from organisation like Tresys Technologies and from the Open Source community.

4.8 Common Criteria for IT Security Evaluation

Organisations would like to implement products which have been certified in order to have the assurance that the products work as specified according to a list of requirements. The use of certified products not only provides the organisation with the assurance that the product works as specified, but also could be a requirement from clients ("*The Common Criteria Evaluation and Validation Scheme*", n.d.). Existing security threats brings the need of consumers of IT products to have confidence in the security features of those products. Governments and other organisations are interested in products which have been evaluated by a non-biased third-party evaluator.

The CC is currently the international standard IS15408 used to evaluate the security of information systems (*"INFOSEC"*, n.d.). Systems are evaluated by assessing the effectiveness and efficiency of their characteristics. A numbered Evaluation Assurance Level (EAL) is granted according to the level of assurance in which the product satisfies the security requirements specified in a document called Security Target (ST). There are 7 levels of evaluation, with EAL1 being the most basic and EAL7 being the most rigorous. The last three levels of evaluation are for those products which were uniquely design to be trusted. Those which initial design was not to be "trusted" systems, but satisfy all the requirements, can get no more than EAL4. The current high water mark is EAL4 with extras. These extras are the plus that can be obtained by evaluating the product against more than one Protection Profile.

In May 2007, RHEL5 client and server were submitted as the Target of Evaluation (TOE) to be certified by the CC. This product was submitted to be evaluated against three Protection Profiles (PP): the CAPP, the RBACPP and the LSPP. In order to comply with the requirements of the RBACPP and the LSPP, the activation of SELinux with the strict policy was necessary.

In June 2007, the CC released the results from the successful certification of RHEL5 at EAL4+ against the CAPP, LSPP, and RBACPP. This certificate was the demonstration that SELinux could provide more than a research tool for organisations. It is important to note that SELinux was evaluated to work properly while implemented in RHEL5. Therefore, SELinux can be considered to work properly only for systems that use RHEL5. Nevertheless, due to similarities between RHEL5 and Fedora Core, it can be assumed that the same results can be obtained from systems that use Fedora Core.

4.9 Access Control Security Goals

Access Control mechanisms have to be implemented in order to prevent or contain damage from security attacks. In HIS, the access controls implemented in the computer system need to be able to address attacks such as masquerading, infiltration, unauthorised disclosure and alteration of information, and DoS attack. In order to mitigate these attacks, security mechanisms in HIS have to address the following security goals:

- Confidentiality and Integrity of data while transmitted.
- Confidentiality and Integrity of data while stored.

- Availability of the information.
- Accountability while working with sensitive data.
- Authentication to accurately verify a claimed identity.
- Non-repudiation.

Most of these goals cannot be satisfied only by access control mechanisms. However, by properly implementing the appropriate access control policy, the security and privacy of the system resources can be maintained and damages mitigated.

SELinux MAC mechanisms can help to address attacks in computer systems and minimise the damage in case an application is compromised. As specified by Herzog, and Guttman (2002) there is a set of goals that can be achieved with SELinux which are:

- Limit raw access to data,
- Protect Kernel integrity,
- Protect system file integrity,
- Separate processes,
- Protect the administrator domain.

SELinux can provide enforcement of least privilege principle and domain separation while protecting system resources. SELinux limits damage from viruses, worms and Trojan horses; inhibits virus propagation; protects against escalation of privileges using Root-kits; and protects against backdoors from insecure applications.

4.10 Open Solutions as a Key Strategy

The use of Open Solutions is becoming more common in the commercial environment to improve the quality of services while reducing implementation costs. Currently the word “open’ is overloaded by different uses and definitions, in this thesis open solutions are defined as those solutions which are based on the public access to the design and content of goods and knowledge following open source definition guidelines. This definition is extended not only to applications’ source code, but also to the standards, knowledge and architectures which are freely available and shared within the open source community.

As specified by Goldstein et al. (2007), Open Solutions are one of the key elements to achieve better healthcare systems. Open Solutions facilitate the ability to communicate and share information between HIS. Goldstein et al. (2007, p. 24) specifies the following open solution elements:

- Open Solutions in the form of open standards.
- Open solutions in the form of OSS.
- Open solutions in the form of architecture.
- Open Solutions in the form of Open Knowledge, Open Innovation and Open Collaboration.
- Open Health and Open Medical.

These elements form the basis for the creation of HIS which improve the quality of care allowing a next-generation of medical informatics and health information technology. The OSS is the most widely implemented of these Open Solutions (Goldstein et al., 2007).

OSS consists of the free distribution of computer software source code for use, modification, improvement and redistribution, while meeting the Open Source Definition guidelines. In these guidelines, users are allowed to have access to the software and its code for any purposes they want, and also they are allowed to redistribute the software to other users.

In the past, organisations have been benefited from the use of OSS. Even if one of the first attractive points about OSS is that it can be used without any cost, it is not the main benefit. Benefits from using OSS come from the fact that it is freely available to everyone. In this way the software code can be used and modified according to specific needs. Also, the software code can be redistributed for further use and/or modification. Benefits can be listed as follows:

- **Reliability.** Making the source code available to a big number of developers help to identify and fix bugs in the code which otherwise will take longer to be discovered and fixed (if discovered at all). Usually a bug could be fixed within hours and a new version or patch is released.
- **Security.** Making the source code available allows for a rigorous inspection and auditing. With a great number of users reviewing the code it is possible to identify potential security bugs or backdoors in the code and correct them.
- **Costs.** Most of the OSS projects are available for free, but the main benefit to the organisations comes from the Total Cost of Ownership (TCO). Low TCO of OSS is the result of possible zero purchase price, low vulnerability to security breaches, no need to account for number of licences used, and possible zero cost of upgrades.

- **Flexibility and Freedom.** Users are able to select a vast variety of software without concern of intercommunication. OSS makes use of open standards and protocols that allow users to select different products which can work according to their requirements. Freedom is achieved when the user is not dependant on a software vendor. If the software vendor runs out of business the user can keep the right to use the software and even make modification over it. OSS projects can also be modified according to the requirements of the user. In case that the user does not have the skills, these requirements can be requested to the Open Source Community.

There is an increasing list of OSS products which are widely accepted by individuals and organisations. Some of the most important products are the Apache HTTP Server, Firefox Web Browser, Open Office, and the Perl language. However, one of the products that best symbolises the OSS products and is considered as the primary OSS OS is Linux. Linux is actually the integration of the Linux Kernel developed by Linus Torvalds and the GNU’s Not Unix (GNU) OS which provides the set of tools that form the complete OS. The GNU project is a complete free system which was integrated in 1992 with the Linux Kernel to create the GNU/Linux OS. This OS is developed under the GNU General Public Licence which allows distributions and sales of possible modified and unmodified versions of the OS but requires all those copies to be released with the same licence.

Linux is becoming one of the most widely used OS threatening MS Windows. Linux has certain characteristics that make it desirable for organisations over proprietary software. Some of these characteristics are its freely availability, mainstream of applications, user-friendly Graphic User Interface (GUI), reliability of code, and secure against most of the viruses and worms. Linux is also becoming one of the first freely available OS which enforces MAC mechanism due to the introduction of SELinux. Linux is becoming the de facto OS to support a variety of servers such as Web servers, printer servers, and Virtual Machines Servers. Also, Linux is moving toward the support for enterprise applications such as Oracle or SAP. According to Gartner, 20% to 25% of users expenditures allocated to Linux will be for mission-critical applications by 2008 (Weiss, 2008).

4.11 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. In addition, SELinux is part of the Open Solutions mentioned by Goldstein et al. (2007), which are needed to provide better HIS in the future. Therefore, SELinux is a recommended viable approach to aid in the protection of resources in HIS.

Chapter 5

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

Contents:

5 RBAC and TE for HIS

5.1 Introduction

SELinux implements flexible and highly granular MAC mechanisms into the Linux kernel which can aid in the protection of the security and privacy of healthcare information. In the last chapter, SELinux was introduced as a viable approach to implement MAC at the OS layer to improve the security in HIS. MAC mechanisms based on TE and RBAC can provide flexibility and granularity while simplifying user management tasks. In addition, loadable policy modules and conditional policies are features in SELinux that can be used to simplify the management of SELinux policies.

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 authorised 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.

5.2 Importance of TE and RBAC

TE in SELinux provides a high level of granularity which can aid in the protection of resources in HIS. HIS are constituted by different types of resources such as data, sockets, processes, etc. In SELinux, every object is labelled with a type and every subject with a domain. Accesses are authorised by comparing the domain of the subject with the type of the object. Also, because TE in SELinux makes use of object classes, the level of granularity while accessing the resources is even higher. The subject has to be authorised to access the combination of type and object class in order to access the resources. Access permissions in SELinux are also different to those existing in traditional Linux systems. The access permissions are not only read, write and execute. Appendix D contains a list of the access permissions existing in SELinux. The great range of access permissions in SELinux increases the granularity while granting access to the resources. In addition, TE in SELinux does not allow any access by default. This means that all access permissions to resources in the system have to be explicitly authorised in order to allow the access. This is a desirable security characteristic that provides a secure by default state in the system.

Access to resources in the system has to be restricted in order to prevent security breaches. However, if a security breach occurs, the damage has to be minimal and the system has to remain secure. In TE, this is achieved by restricting processes to run in domains which are authorised to access only specific objects in the system. The use of TE in SELinux allows the creation of sandboxes in which the processes run. These sandboxes restrict the processes to access only resources inside the boundaries of the sandbox. Therefore, if the process is compromised, the damage is contained within the boundaries of the sandbox.

RBAC helps to minimise the complexity of the user management task in large scale systems. This is achieved by authorising or denying permissions to groups of users based on their role in the organisation. In case permissions are revoked, the system administrator simply has to revoke the permissions to the role and not to individual users. In SELinux, the functionality is very similar. The difference is that in SELinux, access permissions over the resources are not directly authorised to the roles. In SELinux roles are authorised to access a set of domains. In case 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.

The enforcement of the least privilege principle is very important in any computer system, but even more in those that manage sensitive information. Access to resources has to be restricted in such a way that users can only access those resources required to achieve their authorised activities. In SELinux, this is achieved by restricting roles to specific domains. TE is used to create sandboxes to restrict the resources to which processes (domains) running on behalf of users can access. Sandboxes restrict domains to only the minimum amount of access permissions required to achieve their tasks. Consequently, the domains to which the user is authorised to access, according to their authorised roles, determines the privileges that users have over the resources in the system. Thus, restricting the user to the least privileges required to achieve their job functions.

TE and RBAC are a preferred solution over systems that implement MLS. OSs that implement MLS are based on the Bell-LaPadula model, which is mainly dedicated to protect the confidentiality of the information. These systems are considered to be inflexible while protecting information. For example, in a healthcare environment, a physician could be assigned with a *Secret* sensitivity level, while a nurse may be assigned to a C*onfidential* sensitivity level. If the physician writes a diagnostic there is no way in which the nurse would be able to read the diagnostic without compromising the confidentiality of the system (according to the Bell-LaPadula model). Also, MLS does not enforce the principle of least privilege. Let us assume that a user in a *Secret* sensitivity level is authorised to access the *Development* category. This user is able to modify all the resources at the *Secret* sensitivity level in the *Development* category, without regard to the users' authorisation to modify the resource. RBAC and TE in SELinux protect the confidentiality, integrity and availability of the resources along with the enforcement of the least privilege principle.

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.

5.3 SELinux Profiling

First of all, it is important to define profiles and more importantly to define SELinux Profiles. The word "profile" has been used in different contexts for different purposes, hence overloading the term with meanings which could be far from the purposes of this research. The meaning of the word profile, as defined in the *American Pocket Chambers English Dictionary* (1994, p. 524) is:

> "1. a side view, esp. of a face or a head. 2. a brief outline , sketch, assessment, or analysis. – verb trans. (profiled, profiling) to produce a brief assessment of (a subject)."

This definition is not appropriate for the purposes of this research. A preferred definition is of that given in UNIX-based systems. Profile, as defined in the *UNIX Manual Page for profile* (n.d.), is: "setting up an environment for users at login time". Following this definition, and changing it to be more appropriate to the purposes of this research, a SELinux profile is defined as:

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

Figure 4, shows the process followed while assigning a SELinux profile to an authorised 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. This allows the traditional Linux DAC mechanisms to remain orthogonal to the SELinux MAC mechanisms. This also helps to maintain the accountability of actions when users shift to another user or role within their session.

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 authorised 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 authorises *Alice* to access domains (applications) and types (files, executable files) authorised for physicians. SELinux profiles also determine the way in which the domains authorised 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. If the same application is executed by *Bob* who has a SELinux profile for nurses, the application could be restricted to different access permissions over different resources. It is important to mention that SELinux profiles are not necessarily linked with the roles of the organisations, but roles are an important part of SELinux profiles.

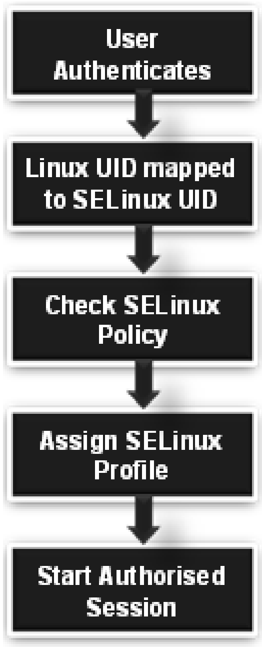


Figure 4. SELinux Profiles Assigning Process.

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. For example, in this research the */healthcare* directory has to exist in all the systems in which the modules are going to be loaded. This directory is labelled with the type *hc_top_dir_t*. This type is used to determine the access permissions that users have over the */healthcare* directory according to their SELinux Profile. In a similar way, executable files such as *diag_sys* (explained further in this chapter) has to be located in the */bin* directory. The executable file is labelled with the type *hc_diag_sys_t*. This type is used to determine the domains which can access the file and the ones who have entry point permission. If directories, files, executables and so on are not in the expected path, the SELinux Profiles would not have effect in the system resources. The directories tree structure and executable files created in this research can be found in the *file context* file found in Appendix B. The *file context* file of the loadable policy module determines the default security contexts for these resources.

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 labelled when the system starts, if changes to the module require modification to the security contexts of any object, the objects need to be relabelled. 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 5 shows the steps that a SELinux policy administrator has to follow when managing loadable policy modules.

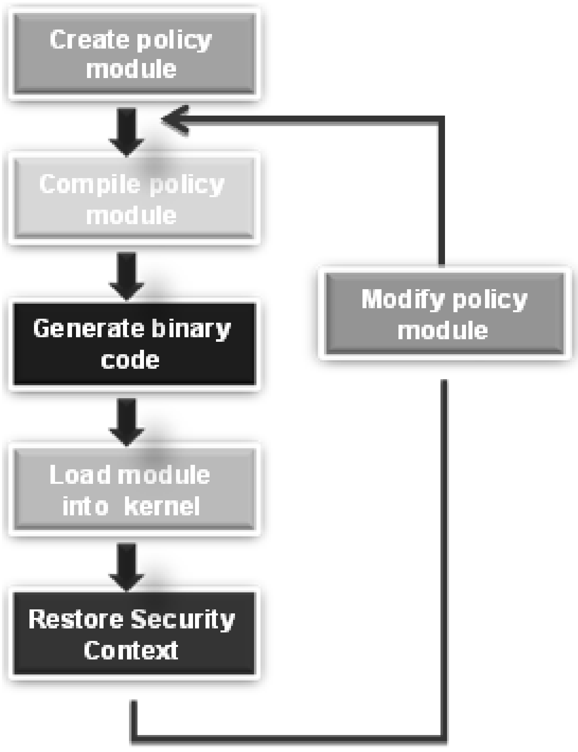


Figure 5. Loadable Policy Modules Management Process

The following sections describe the MAC mechanisms along with the Linux DAC mechanisms that form part of SELinux Profiles. SELinux profiles are mainly based on flexible SELinux MAC mechanisms. However, traditional Linux DAC mechanisms also form an important part of SELinux Profiles.

5.3.1 DAC Mechanisms in SELinux Profiles

SELinux was implemented following the LSM framework, meaning that traditional Linux DAC checks have to be passed before the MAC mechanisms in SELinux are called. DAC mechanisms in Linux provide an extra level of security while protecting resources managed by the OS. The DAC mechanisms in Linux are based on the UID and GID of the subjects (processes) and the permission bits of the objects (files, directories).

In Linux systems, every user is given a unique number which is the UID. Also, a user can belong to one or more groups, each represented by a unique number which is the GID. Once the user is authenticated and assigned to a UID and GIDs, all processes run by the user acquire his/her UID and GID.

Access to objects in Linux systems is restricted by the permission bits of the object. There are basically three permissions that can be applied to every object in the system, which are: read, write and execute. If a permission is not set, the rights to permission grants are denied. Permissions are managed in three distinct classes, which are: owner, group and others. The most common way to represent permission bits is as follows:

“rwxr-xr-x”

In this symbolic representation, each class of permissions (owner, group and others) is represented by three characters. The first set of three characters represents the owner, the second set represents the group and the last set represents the others. The set of three characters represents the read, write and execute permissions over the object. For example, in the set of permissions for the group, which are “r-w”, those who belong to the group of the object can read and execute but not write on it.

The subject’s UID and GID are compared against the permission bits of the object to determine authorised access permissions. For example, if the set of permission bits for the groups in a file object are “r-x”, if the file has a GID of 5001 and the process has the same GID, then the process can read and execute the object but not write on it.

The DAC mechanisms in Linux have to be managed properly in order to be effective. That is, these mechanisms have to restrict the read, write and execute permissions over the system resources to specific users and/or groups of users. However, it is important to remember that in DAC systems once a user has access to the resources the OS cannot restrict the discretion in which users can pass access permissions to others.

5.3.2 Flexible MAC Mechanisms in SELinux Profiles

DAC mechanisms do not fully satisfy security requirements in computer systems. Access to resources has to be restricted according to the security policy implemented in the OS. SELinux Profiles assign users to specific roles which determine the domains authorised for the users. That is, users are restricted to run certain applications according to their role in the organisation. In this way, users can be limited to the least number of privileges over the resources in the system by limiting the number of applications to which they can access.

Applications running on behalf of the users have to be restricted to specific system resources in order to avoid any intentional or unintentional damage. SELinux Profiles restrict the environment of applications running on behalf of the users. With SELinux Profiles, the applications run inside sandboxes, restricting the number of resources the application can access to those inside the boundaries of the sandbox. Consequently, if the application is compromised, the damage would be restricted to the boundaries of the sandbox. Sandboxes are created using domains and TE rules.

SELinux Profiles are constituted by the following SELinux MAC mechanisms:

- **Orthogonal User Identifiers.** User identifiers in SELinux are orthogonal to traditional Linux user identifiers. In this way, SELinux MAC mechanisms remain orthogonal to the Linux DAC mechanisms.
- **Type Enforcement.** Every subject and object in the system is labelled with a type, which is used for access control decisions made by the SELinux Security Server. The use of TE allows a high level of granularity, while restricting access to resources since no access is allowed by default. TE allows the implementation of domain separation through the use of transition rules. Domains separation is used to create sandboxes in which applications running on behalf of the user operate.
- **Role-Based Access Control.** Roles are used to simplify the user administration task and to enforce the principle of least privileges.

5.3.3 Booleans in SELinux Profiles

Conditional policies are an important part of SELinux Profiles. Conditional policies are used to grant or revoke authorised access permissions to subjects in specific circumstances. Two types of booleans can be used in SELinux Profiles, which are (Mayer et al., 2007):

- **Tuneable Booleans.** These Booleans are those that can be changed at run-time, enabling and disabling pieces of the SELinux policy.
- **Persistent Booleans.** This Booleans are those that can remain static once the policy is loaded into the kernel until the entire policy is loaded again, that is, at the next system boot.

Persistent Booleans are used by SELinux profiles to determine changes in authorised access permissions depending on the type of system in which the SELinux Profile is implemented. TE rules in SELinux Profiles can be enabled or disabled according to the Linux distribution or the type of computer used. For example, the security policy could require the user of a laptop to have different access permissions to system resources than the user of a Personal Computer (PC). SELinux Profiles supports this type of change through the use of Persistent Booleans. The value of the Persistent Boolean is used to determine authorised accesses in SELinux Profiles, which are set when the policy is loaded into the kernel at system boot. These changes remain static every time the system reboots.

Tuneable Booleans are used by SELinux Profiles to determine changes to authorised access permissions, which have to be done at run-time depending on special circumstances. The use of Tuneable Booleans allows enabling or disabling AVRs in SELinux Profiles at run-time without the need for modifying the code and having to reload the policy module into the kernel. For example, in HIS, changes to the SELinux policy would have to be made in emergency situations, privileges would have to be granted to specific users, enabling them access rights to restricted resources. In an emergency scenario, Tuneable Booleans can be used to enable AVRs in the SELinux Profiles, authorising specific users or roles to access objects in the system. These changes can be made on the spot without the need to reboot the OS or reload the policy module (corresponding to the SELinux Profile).

Nevertheless, special care has to be taken while creating SELinux Profiles that enable or disable access permissions based on Tuneable Booleans. Tuneable Booleans directly impact the security policy. If an attacker has access to the files that contain the values of the Tuneable Booleans, an attacker could authorise access permissions, leading to an unauthorised use or disclosure of resources. Therefore, it is recommended that only the minimum amount of access permissions are impacted by the Tuneable Booleans. Also, only specific subjects should be authorised to access the files containing the values of booleans. This research recommends the creation of a special user which is only dedicated to this task (even apart from the system administrator user).

5.4 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 and the way in which Tuneable Booleans can be used. This scenario is one of the different scenarios used to test the prototype that was developed as proof of concept for this research. A simple healthcare application called the *Diagnostic Application* and related files and directories were created as part of this scenario. The policy module for the *Diagnostic Application* can be found in Appendix B. This module contains the access permissions part of the SELinux Profile assigned to users so that they can interact with the *Diagnostic Application*. The content of the code found in Appendix B is not explained in detail. In order to understand more about how to create SELinux Policies, it is recommended to consult the book: “SELinux by Example” by Mayer et al. (2003).

SELinux Profiles as described in this thesis, have certain requirements in regards to tools and technologies that need to be present in the system in order for a successful deployment. This section also introduces tools and technologies which allow replicating the described scenario in other systems.

5.4.1 System Requirements

The development and implementation of the policy modules used in this research require certain tools and technologies. The tools and technologies used here are the ones that were available at the time this research was made, but it is likely that in the future more advanced and useful tools and technologies would be released. SELinux is changing constantly and it is expected that future tools and technologies will streamline its management.

In order to replicate the results of this research, it is required and recommended to use and implement the following tools and technologies:

Reference Policy

Tresys Technology is responsible for the development of the Reference Policy project which has the main goal of restructuring the NSA example policy for SELinux ("*SELinux Reference Policy*", n.d.). The Reference Policy is based on two main design principles: Layering and Modularity.

Currently, the Reference Policy is constituted by five different layers which are: Apps (applications), Kernel, System, Admin (administration), and Services. 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 *Diagnostic Application* are placed on the Apps layer since it has its own domain types. That is, the *Diagnostic Application* is not directly related with the kernel or any other system services.

Modularity is achieved through the use of encapsulation and abstraction. Encapsulation is used to keep types and attributes private to the module in which they belong. For example, all types and attributes related to the *Diagnostic Application* remain private to the modules of the *Diagnostic Application*. Abstraction helps to group pieces of code so that it can be accessed from inside or outside of the module. There are two types of abstractions: interface and templates. Interfaces are the most commonly used, allowing access to private components between modules. Templates are used to create types in external modules but the functionality remains private to the external module.

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.

Appendix B and C provide an example of these files.

Linux Distribution

Since their version 4, Fedora and RHEL were released with SELinux as a fully enabled by default security feature. Other Linux distributions (Gentoo, Debian, Ubuntu, CentOS and Engarde) can be customised to work with SELinux. However, making it work properly is not a simple task. For the simplicity and native implementation of SELinux, Fedora Core or RHEL are the recommended Linux distributions to use. It is also recommended to use the latest versions, since these versions make use of the Reference Policy and loadable policy modules. Also, latest versions of Fedora Core provide command line tools which simplify the management of SELinux policies, such as *semanage*. Fedora Core 8 was the last version of Fedora Core released during the development of this prototype, thus it was the version used. However, once this research was finished, Fedora Core 9 was released which also works according to the specifications described in this thesis.

Targeted Policy

In order to simplify the task of a system administrator while managing SELinux polices, Red-Hat developed two SELinux policies. These SELinux policies are called: the targeted and the strict SELinux policies. The Targeted SELinux policy basically restricts twelve specific domains (i.e. apache daemon), placing all other domains under the *unconfined_t* domain. In this way the Linux system operates as normal however, it is restricting authorised accesses for those 12 domains (Bauer, 2007). The strict SELinux policy enforces the principle of TE in which no access is allowed by default. However, this increases its complexity and requires manual customisation for correct operation of the system. For simplicity purposes, the Targeted Policy was used in this research in order to provide the base-modules to interact with the non-based modules created.

SELinux Policy Development Package

For the development of SELinux policies it is important to install the SElinux policy development package. Currently the most recent version for Fedora Core 8 is "selinux-policy-devel-3.0.8" which requires the following packages: libselinux and selinux-policy. This package was used to compile the policy modules created in this research.

SELinux Policy Development Tools

Tresys Technology provides a set of tools to simplify the analysis, development, and testing of SELinux policies. Two of these tools were used during the development of this research, which are:

- **SELinux Policy IDE (SLIDE).** Integrated Development Environment for SELinux which is delivered as a plug-in for Eclipse SDK. Its GUI makes the task of developing SELinux policies easy due to its policy syntax highlighting and auto-completion capabilities. Also, SLIDE allows the easy use of compiling and building module packages.
- **SETools.** These are open source tools for the analysis of SELinux policies. From this set of tools, mainly Apol was used for the analysis of SELinux policies.

5.4.2 The Diagnostic Application

The Diagnostic Application has the following requirements:

The diagnostic application allows creating diagnostic reports for specific patients. Once the information of the diagnostic report is provided, it is stored in two separated files with similar names (diagnosis.di) but in different directories and with different contents. The file stored in the directory belonging to the patient contains personal details of the patient. The other file which is stored in the directory that corresponds to the researchers contains only statistical data which cannot be related to a specific patient.

Physicians are authorised to read the diagnostic reports that they created for specific patients. These diagnostic reports can be checked by the patient so that they can read the symptoms and opinions given by the doctor. Nurses are also allowed to check the diagnostic belonging to specific patients, so that they can appropriately treat them. Researchers are authorised to have access to the diagnostic reports for statistical purposes, but they cannot access information that is related to specific patients.

In this application the only users allowed to create and delete diagnostic reports are the physicians. Physicians, nurses and patients can access the information in the files stored in the directory for patients. Researchers can only access the information inside the files located in the directory for researchers.

All these operations have to be made through the use of the Diagnostic Application. Users are not authorised to access the patient and researcher directories, along with the files inside these directories, by any other means.

In addition, in case of emergency the diagnostic reports are allowed to be checked by a local general practitioner (GP). In normal situations the local GP is not able to access any of these reports.

5.4.3 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 *Diagnostic Application*. The identified roles are described in Table 4.

Role	SELinux Role	Description
Doctor/ Physician	hc_doc_r	Physicians are authorised to create, check and delete diagnostic reports. This includes reading, writing, appending and deleting files in the patients' directories.
Nurse	hc_nur_r	Nurses are authorised only to check diagnostic reports. This includes only the reading of files in the patients' directories.
Patient	hc_pnt_r	Patients are authorised only to check their diagnostic report. This includes reading files only from their directory.
Researcher	hc_res_r	The researchers can check diagnostic reports that cannot be related with the patients. This includes reading files in the researchers' directory.
Local GP	hc_locgp_r	The local GP can only check diagnostic reports in emergency circumstances. This includes reading files from the patients' directories only in case of an emergency.

Table 4. Diagnostic System Roles

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

5.4.4 Role Based Access Control and Type Enforcement

Once the roles are defined, the next step is to determine the domains authorised 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 *Diagnostic Application* are shown in Table 5.

Object (Resource)	Object Class	Type
/healthcare/db/patient/<patient_name>	dir	hc_pnt_dbdir_usr_t
/healthcare/db/researchers	dir	hc_res_dbdir_t
/healthcare/db/patient/*/diagnostic.di	file	hc_pnt_dbfile_di_t
/healthcare/db/researchers/diagnostic.di	file	hc_res_dbfile_di_t
/bin/diag_sys	file	hc_diag_sys_exec_t

Table 5. Diagnostic Application Types

The first two types shown in Table 5 are the ones corresponding to the patients and researchers directories, in which the *Diagnostic Application* stores the diagnostic reports. The following two are the files containing the data from diagnostic reports for patients and researchers. The last one is the type corresponding to the executable file of the *Diagnostic Application*. Appendix B has the code required to set these types as part of the security contexts in the directories and files of the *Diagnostic Application.* This code is found in the *diag_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 authorised to all roles. This domain could be the one that corresponds to the *Diagnostic Application* process. However, this is not a recommended approach. If only one domain is used for the *Diagnostic Application* process, all the roles authorised to access this domain would have the same privileges while using the *Diagnostic Application*. In this way the least principle privilege is not enforced. If a researcher is able to compromise the code, the researcher would be able to access objects that the doctors, patients and nurses 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 *Diagnostic Application* 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 researcher would not be able to access resources that are not authorised for that role while using the *Diagnostic Application*. Table 6 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**
hc_doc_r	hc_doc_diag_t	hc_pnt_dbdir_usr_t	add_entry_dir_perms create_dir_perms search_dir_perms
		hc_pnt_dbfile_di_t	create_file_perms rw_file_perms read_file_perms
		hc_res_dbdir_t	create_dir_perms add_entry_dir_perms
		hc_res_dbfile_di_t	create_file_perms rw_file_perms
hc_nur_r	hc_nur_diag_t	hc_pnt_dbdir_usr_t	search_dir_perms
		hc_pnt_dbfile_di_t	read_file_perms
hc_pnt_r	hc_pnt_diag_t	hc_pnt_dbdir_usr_t	search_dir_perms
		hc_pnt_dbfile_di_t	read_file_perms
hc_res_r	hc_res_diag_t	hc_res_dbdir_t	search_dir_perms
		hc_res_dbfile_di_t	read_file_perms

hc_locgp_r	hc_locgp_diag_t	hc_pnt_dbdir_usr_t	search_dir_perms
		hc_pnt_dbfile_di_t	read_file_perms

Table 6. Diagnostic Application Roles, Domains and Types

A user that is authorised to access the role *hc_res_r* (researcher) is subsequently authorised to access the domain *hc_res_diag_t* which is assigned to the *Diagnostic Application* process. Consequently, this user is authorised only to *search_dir_perms* in the researchers' directory and *read_file_perms* the files with the name diagnosis.di inside this directory. Information about access permissions can be found in Appendix D.

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

Nevertheless, the first domain which the users' role has to be authorised 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 7 shows the shell process domains associated to the roles used in the *Diagnostic Application* scenario.

Role	Shell Domain
hc_doc_r	hc_doc_t
hc_res_r	hc_res_t
hc_nur_r	hc_nur_t
hc_pnt_r	hc_pnt_t

Table 7. Linux Shell Types

These types have to be authorised to transit into the domains specified in Table 6. This has to be done in order to authorise the shell process running on behalf of the user, to enter the corresponding sandbox. That is, the domain *hc_doc_t* has to be authorised to transit into the domain *hc_doc_diag_t*, and the domain hc_pnt_t has to be authorised to transit into the domain *hc_pnt_diag_t* and so on.

5.4.5 Conditional Policies

In Table 6, the local GP role is authorised to access patients' directories and files. These access permissions are unauthorised in normal circumstances. However, in an emergency situation these permissions have to be enabled so that the local GP role can access the resources. In order to enable or disable access permissions, a conditional statement and a boolean called *hc_diag_emerg* are used in the policy module code. The code in Appendix B shows how this boolean was created and used in the policy module. Also, it shows how conditional statements were created to enable or disable pieces of the SELinux policy.

Booleans are created in the policy modules using the following statement:

```
bool <boolean_variable>;
```

TE rules are contained within a conditional statement which makes use of a boolean to determine if the TE rules have to be enabled of disabled. The conditional statement syntax is as follows:

```
If (<boolean_variable>){

#Rules to be enabled or disabled

}
```

5.4.6 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 authorise 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 researcher executes the *diag_sys* executable file. That is, the domain *hc_res_t* by default transits to the *hc_res_diag_t* domain when executing a file with type *hc_diag_sys_exec_t.*

```
type_transition hc_res_t hc_diag_sys_exec_t : file hc_res_diag_t
```

- **AVRs.** The following AVRs authorise the *Diagnostic Application* process running on behalf of a researcher to *search_dir_perms* in the researchers' directory and to read files inside the directory. That is, the domain *hc_res_diag_t* is authorised to search directories with the type *hc_res_dbdir_t* and read files with type *hc_res_dbfile_di_t*.

```
allow hc_res_diag_t hc_res_dbdir_t:dir { search_dir_perms };

allow hc_res_diag_t hc_res_dbfile_di_t:file { read_file_perms };
```

- **RBAC rules.** The following role statement authorises researchers to access the shell process and the *Diagnostic Application* process. That is, the role *hc_res_r* is authorised to coexist with the domains *hc_res_t* and *hc_res_diag_t* in the security context of a subject (process).

```
role hc_res_r types { hc_res_t hc_res_diag_t };
```

Along with these rules, it is required to specify the AVRs for domain transitions as follows:

```
allow hc_res_t hc_diag_sys_exec_t:file { getattr read execute };

allow hc_res_diag_t hc_diag_sys_exec_t:file entrypoint;

allow hc_res_t hc_res_diag_t:process transition;
```

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

In Appendix B, all of the other TE rules and RBAC rules required to authorise the access permissions specified in Table 6 are specified within the policy module.

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/devel/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 <module_name>.pp
```

After the module is successfully loaded into the kernel, it will provide the roles, types, domains, TE rules, RBAC rules and booleans that form part of the SELinux Profiles. For example, once the policy module in Appendix B is loaded into the kernel, it will provide the SELinux Profiles for users of the *Diagnostic Application*.

However, before the SELinux Profiles can take effect over the resources used by the *Diagnostic Application*, the security context 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 ]# restorecon –r –v <path_to_files>
```

5.4.7 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. This is done with the *semanage* command as follows:

```
[root@host ~]# semanage user –a –R “hc_res_r” –P user hc_res_u
```

This command creates a SELinux UIDs *hc_res_u* which is authorised to access the role *hc_res_r*.

Now the system administrator can create Linux UIDs which can be mapped to the SELinux UIDs. There are two ways of doing this. If the Linux UID has already been created, the system administrator can use the *semanage* tool to do the mapping as follows:

```
[root@host ~]# semanage login –a –s hc_res_u bob
```

Or, if the Linux UID has not been previously created, the system administrator can use the *adduser* command as follows:

```
[root@host ~]# adduser –s /bin/bash –d /home/bob –c “Bob” –Z hc_res_u bob
```

In any case, once this is completed the Linux UID *bob* is mapped to the SELinux UID *hc_res_u*.

This is the final step in the creation and implementation of the loadable policy modules to create SELinux Profiles that restrict access permissions of the *Diagnostic Application* users.

5.4.8 A closer look at SELinux Profiles

Let's assume that a physician called *Alice* is authorised to access the *Diagnostic Application*. The physician *Alice* will have the following identifiers:

Linux UID: dralice

SELinux UID: hc_doc_u

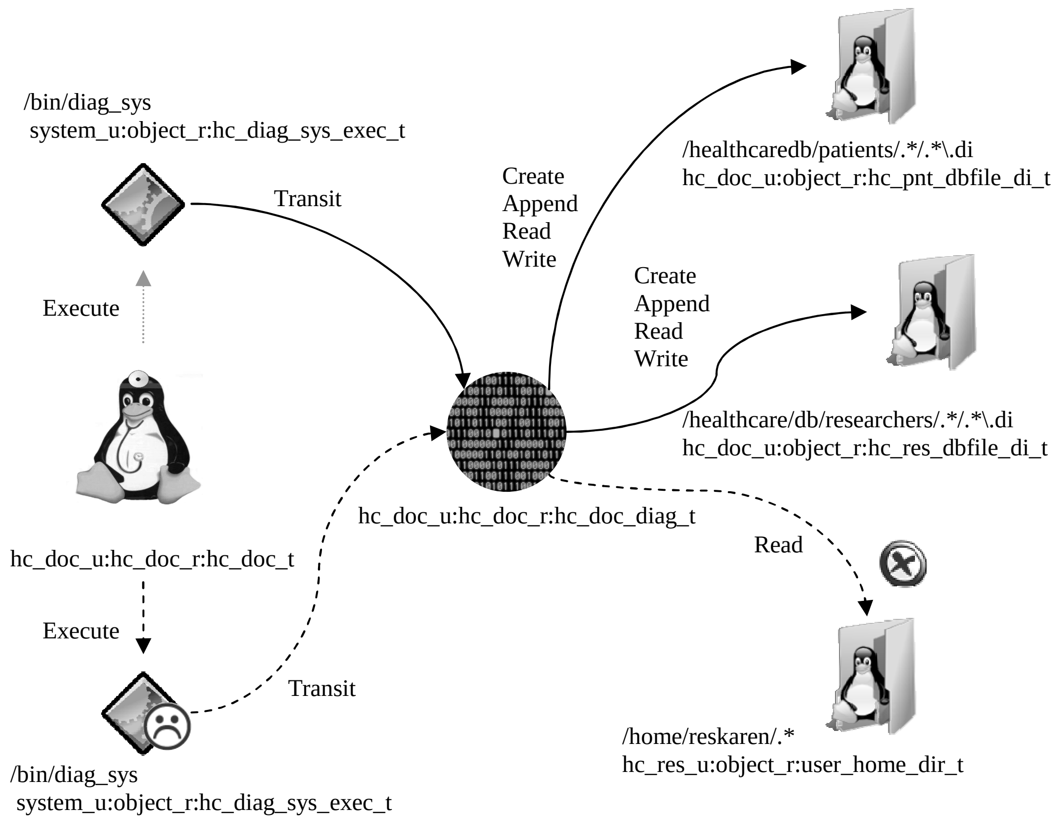
Authorised access permissions are as stated in Table 6 and Table 7, and roles are those stated in Table 4. In addition, the SELinux UID *hc_doc_u* is authorised to access the role *hc_doc_r*.

Following the process shown in Figure 4, once the user logs in, the user is assigned to the Linux UID *dralice*, which is mapped to the SELinux UID *hc_doc_u*. The SELinux UID is used to determine the SELinux Profile that corresponds to *Alice*. In regards to the *Diagnostic Application,* the SELinux Profile of *Alice* is constituted by the roles *hc_doc_r,* the domain *hc_doc_diag_t* and the types *hc_pnt_dbdir_usr_t, hc_pnt_dbfile_di_t, hc_res_dbdir_t, and hc_res_dbfile_di_t*. Also, the SELinux profile is constituted by the access permissions that the domain *hc_doc_diag_t* has over each of the types.

The SELinux Profile of *Alice* authorises the creation of diagnostic reports through the use of the *Diagnostic Application*. This is authorised because the role *hc_doc_r* can access the domain *hc_doc_diag_t*. This domain has the access permissions, create and write over files with type *hc_pnt_dbfile_di_t* (this type corresponds to the files containing the diagnostic reports of the patient) and create permission over the directories with type *hc_pnt_dbdir_usr_t* (which corresponds to the directory of the patient). The domain *hc_doc_diag_t* also has the access permissions, create and write over the files with type *hc_res_dbfile_di_t* and directories with type *hc_res_dbdir_t.*

If an attacker modifies the binary of the *Diagnostic Application* and attaches a backdoor in the code, the attacker could try to access sensitive information of the user. For example, the attacker could try to read information in the user's home directory. However, because of the restrictions set by the SELinux Profile, this attack is not possible. As shown in Figure 7, even if *Alice* executes the bogus *Diagnostic Application*, the application does not have access to the resources in *Alice's* home directory. The SELinux Profile of *Alice* restricts the domain *hc_doc_diag_t,* which is the *Diagnostic*

Application process running on behalf of a physician, to specific object types. The object types that can be accessed by the domain *hc_doc_diag_t* do not include the type *user_home_dir_t*. Consequently, the SELinux Profile of *Alice* makes it impossible for an attacker to access object types that are not specified in the SELinux Profile.



/bin/diag_sys
system_u:object_r:hc_diag_sys_exec_t
Transit
Execute
Create
Append
Read
Write
/healthcaredb/patients/.*/.*\.di
hc_doc_u:object_r:hc_pnt_dbfile_di_t
Create
Append
Read
Write
/healthcare/db/researchers/.*/.*\.di
hc_doc_u:object_r:hc_res_dbfile_di_t
hc_doc_u:hc_doc_r:hc_doc_diag_t
Read
hc_doc_u:hc_doc_r:hc_doc_t
Execute
Transit
/home/reskaren/.*
hc_res_u:object_r:user_home_dir_t
/bin/diag_sys
system_u:object_r:hc_diag_sys_exec_t

Figure 6. SELinux Profiles for Diagnostic Application Physicians

In order to explain how SELinux Profiles work during an emergency situation and following the *Diagnostic Application* scenario, a local GP *Bob* has to be authorised to access the diagnostic reports of a specific patient during such a situation. The user *Bob* is assigned to the following UIDs:

Linux UID: locgpbob

SELinux UID: hc_locgp_u

The user *Bob*, in normal conditions is not authorised to access any resource in the system. In an emergency situation however, the local GP has to be given the authorisation to read the diagnostic reports required of specific patients. The SELinux Profile of the user *Bob* authorises the SELinux UID *hc_locgp_u* to access the role *hc_locgp_r*. As stated in Table 6, this role is authorised to access the domain *hc_locgp_diag_t*. However, all the TE rules that authorise the domain *hc_locgp_diag_t* to access any objects are disabled by default. The TE rules that authorise these accesses are contained within a conditional statement and are subject to the value of a Tuneable Boolean called *hc_diag_emerg*.

The use of the *hc_diag_*emerg Tuneable Boolean allows the authorising of access permissions to the user *Bob*, without the need of recompiling and reloading the SELinux policy. Also, is important to mention that the use of *auditallow* rules in conditional policies help to keep track of all the actions made by *Bob* once the access permission are authorised. Therefore, if *Bob* takes advantage of his privileges, the damage made can be tracked and *Bob* can be prosecuted. Figure 5 exemplifies the behaviour of how the conditional policies work with the SELinux Profiles. If the value of the *hc_diag_emerg* Boolean is *false*, read access to the files with type *hc_pnt_dbfile_di_t* is denied. But if the value of the *hc_diag_emerg* Boolean is *true*, read access to files with type *hc_pnt_dbfile_di_t* is authorised.

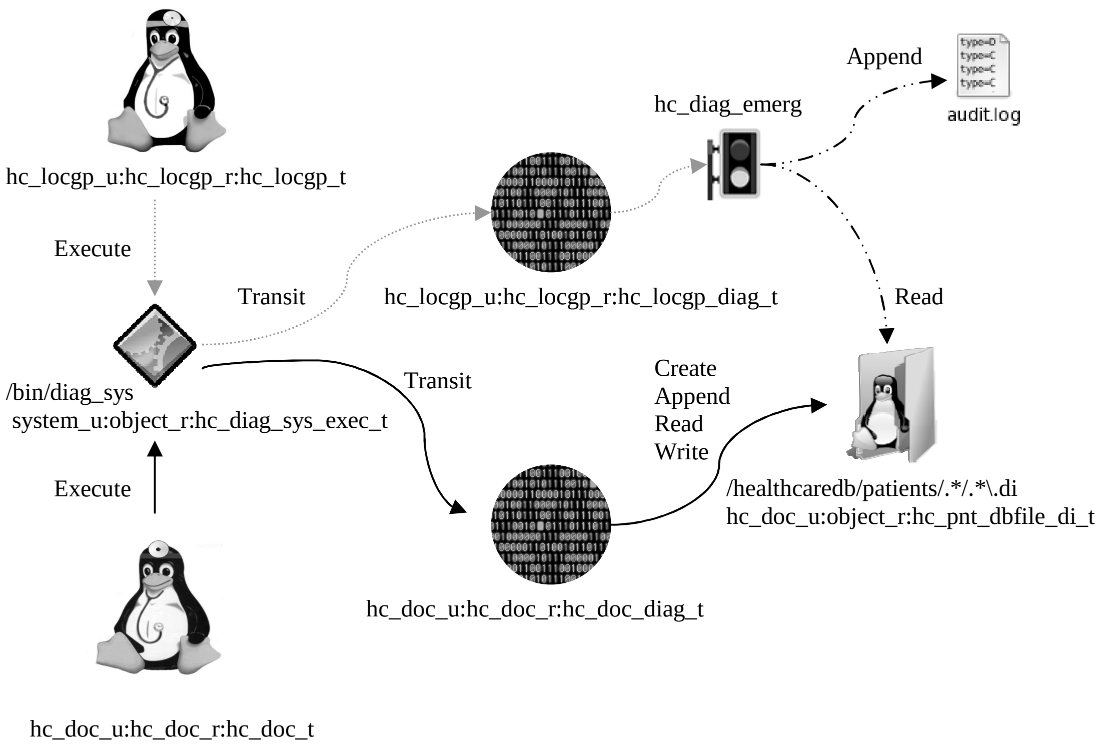

hc_locgp_u:hc_locgp_r:hc_locgp_t
Execute
Transit
hc_locgp_u:hc_locgp_r:hc_locgp_diag_t
hc_diag_emerg
Append
audit.log
Read
/bin/diag_sys
system_u:object_r:hc_diag_sys_exec_t
Transit
Create
Append
Read
Write
/healthcaredb/patients/.*/.*\.di
hc_doc_u:object_r:hc_pnt_dbfile_di_t
Execute
hc_doc_u:hc_doc_r:hc_doc_diag_t
hc_doc_u:hc_doc_r:hc_doc_t

Figure 7. Conditional Policies and SELinux Profiles

The value of booleans can be checked using the *getsebool* tool as in the following statement.

[root@host ~]# getsebool hc_diag_emerg

The value of booleans can be changed in two ways. One is with the use of the *setsebool* tool as follows:

```
[root@Orca1 ~]# setsebool hc_diag_emerg true
[root@Orca1 ~]# getsebool hc_diag_emerg
```

The second one is to directly modify the files in the SELinux pseudo filesystem (typically mounted on /selinux)

```
 [root@Orca1 ~]# echo 1 >> /selinux/booleans/ hc_diag_emerg
[root@Orca1 ~]# echo 1 >> /selinux/commit_pending_bools
```

Both statements set the value of the *hc_diag_emerg* Boolean to be *True*. Therefore, the conditional statement in the policy module enables the TE rules that authorise the local GP user to access the files with type *hc_pnt_dbfile_di_t*.

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 unauthorised 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 authorised 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. Also, the use of conditional policies was demonstrated to be a useful solution during emergency situations.

Chapter 6

Application Layer Security

Contents:

6 Application Layer Security

6.1 Introduction

Systems are constructed with the idea that security can be provided at the application layer and that systems can be protected using firewalls, antivirus software, and so on. However, what could happen if an attacker is able to bypass access rules in the firewalls or if the antivirus does not have the fingerprint of a new virus? What could happen if system administrators do not patch the system's applications in a constant and methodical way? What could happen if by compromising an application the attacker is able to change the way in which users access the organisation's DB?

Most of these issues will lead to an unauthorised disclosure or modification of the information. Access control has to be provided in the computer system so that only authorised users have access to the information. However, even if the application enforces access control rules, these could be bypassed. In complex applications with thousand of lines of code, there is a high probability of code errors which could lead to compromise the application and even the entire system. In order to provide an in-depth security solution in computer systems, it is required the support from the underlying OS. MAC mechanisms in the OS can help to prevent attacks or minimise the damage from compromised applications. SELinux has been demonstrated to be a viable solution to enforce a flexible MAC in HIS.

In this chapter different vulnerabilities at the application layer are explained with past experiences in which these vulnerabilities have been exploited by an attacker. Virus, worms and trojans have been a threat in computer systems for decades. However, they are still compromising the security and privacy of the information due to its possibility to escalate privileges in OS that implement DAC mechanisms. This chapter describes how SELinux can prevent these types of attacks and help to minimise the damage from compromised applications.

6.2 The Inevitability of Software Failure

One of the most important aspects while creating computer systems is to understand the importance of the OS to support security mechanisms at the application layer. Loscocco et al. (1998) stated that computer systems are constructed with "the flawed assumption that security can adequately be provided in application space without the support from the underlying Operating System". It has been almost a decade since this statement was published, but systems are still compromised when an attacker finds the way to exploit vulnerabilities in application layer software. Applications are very complex pieces of software which include thousands of lines of code. Within these lines of code exists some vulnerability and it is just a matter of time before it is discovered by both, the vendor or the attackers.

A clear example of the concern about the vulnerabilities in applications in these days is the emphasis on the deployment of firewalls, IDSs, antivirus, and other types of defence to protect systems for attacks. These defence mechanisms are deployed because the great number of vulnerabilities in much of the applications used in computer systems. However, these defence mechanisms are also pieces of software which due to a wrong configuration or own vulnerabilities could be bypassed and other vulnerabilities in the system exploited.

Viruses, worms, buffer overflows, etc. are common types of malware to which applications are continuously susceptible. Compromised systems by malicious code are continually announced in security news and cost billions of dollars per year to organisations around the world (Daswani et al., 2007; Sahadi, 2005; Ledney, 2008). These types of attacks are due to applications that are flawed and there is always an attacker looking to exploit these vulnerabilities.

6.3 Application Layer Security Issues

Application software is prone to be flawed and consequently the resources to which the user is allowed to access are in risk of being compromised. In a computer system access to the resources is granted to the users based on the new-to-know over specific resources. However, users are not the ones who directly access the resources, that is, users do not "digitalise" themselves to access files in a computer system. Users make use of applications in the computer systems to access the resources. Therefore, these applications are granted with all the privileges that users have over the computer system. In DAC this fact means that the resources which a user is allowed to access are on discretion of the applications the user is running.

There is a great effort to make application software more secure and reliable, but no matter how much effort is spent, application software will always be flawed. Initiatives from office applications, like MS Office, or those from advanced DB servers, like Oracle and MS SQL server, have demonstrated that security requirements are important for software vendors but they would never be completely satisfied. In this section some of the most commonly implemented products in computer systems are introduced with their security features and vulnerabilities in order to demonstrate the consequences of failures at the application layer.

6.3.1 Microsoft Office

There have been an increasing number of attacks while opening MS office documents received through e-mail or downloaded from web sites in the Internet (Hoffman, 2008; McMillan, 2007; McMillan, 2006). These attacks take advantage of vulnerabilities in MS Office leading to memory corruption or buffer overflow allowing the attacker to escalate privileges and compromise the users' system. If necessary, MS provides security updates for their products every Tuesday of each month and publishes bulletins to announce the updates. However, because users do not update their products or due to the speed with which attackers find other vulnerabilities, attackers are still able to compromise systems using malware. One of the most vulnerable applications is MS Excel, which represent a high risk due to its extensive use in all types of organisations. An example of this, in January 2008, MS warns about vulnerability in MS Excel 2003 which allows an attacker to execute remote code in a Windows or Mac system (Kirk, 2008).

6.3.2 Web Browsers

Internet Explorer (IE) has been one of the most common web browsers for years. IE has been considered to have serious vulnerabilities in all its released versions (Sayer, 2006; Krebs, 2006; McWilliams, 2002). These vulnerabilities are the result of design flaws, bugs and third party applications. An example of third party vulnerability is the introduction of java script in web pages or the use of Active-X. The last version of IE, version 7, has demonstrated a considerable increase on security features. IE7 includes security features such as a Phishing filter, SSL alerts, Active-X opt-in, and a "containment wall". This last one is one of the most useful features preventing the web browser from installing software or changing computer setting without the consent of the user. However, not even 24 hours after the release of IE7 the first information disclosure vulnerability was reported (Sayer, 2006).

Currently, Firefox is becoming one of the most commonly used web browsers after Microsoft IE. Firefox is developed with the support of the open source community and Mozilla Corporation. As with any other open source project, making the source code available to everyone allows it to detect and solve vulnerabilities faster than proprietary code. However, changing from IE to Firefox does not imply that users are more secure, security vulnerabilities also exist in Firefox (Popa, 2007).

6.3.3 Databases

DBs are created with the purpose to store huge amounts of information and make it available as fast and reliably as possible. Because of the importance of the data stored in

DB systems, there is a need to increase the security mechanisms in order to protect the data against unauthorised outsiders or illegal insiders attempting to exceed their authority. These mechanisms have varied through the years from simple password protection like in MS Access, to complex user/role structures supported by advanced DBs like Oracle or MS SQL server. DBs are very complex pieces of software which are prone to code errors which could allow an attacker to compromise the whole system. DBs are prone to attacks such as SQL injection attacks in which requests made by the clients through the web server could not be properly analysed and could result in arbitrary commands to be executed in the DB. Even with the increasing efforts from different organisations, SQL injection attacks are still happening. An example is explained by Fisher, (2008) in which Oracle DBs can be forced to execute the command of the attacker. DBs are also susceptible to worms and viruses. The SQL Slammer worm released in 2003 was dedicated to exploit the buffer overflow vulnerability in MS SQL servers. The worm was able to infect 75,000 hosts, 90 percent of them in the first 10 minutes (Daswani et al., 2007).

6.3.4 Web Servers

These days web servers are one of the most important parts of organisations' computer systems, making information available over the Internet. Nevertheless, web servers are the most frequently targeted hosts while compromising computer systems. Viruses, worms, and Trojans are common attacks to which web servers are vulnerable. Once an attacker gains unauthorised access to the server, the attacker is able to change information on the site (defacement attack), access sensitive information, or install malicious software for further attacks (Rootkit). Also, an attacker can commit a DoS attack impeding authorised users to access organisations' websites.

The most commonly implemented web server is Apache Web Server followed by MS IIS. Apache is one of the web servers who played a key role during the initial growth of the World Wide Web. Even if Apache web server is considered to be secure, there are security issues which could lead to breaches if the web server is not properly configured or patched ("Apache Tomcat", n.d.). Inappropriate user permissions could allow access to sensitive files to malicious users, faulty CGI and PHP scripts can be used to compromise the system, and FrontPage extensions could be insecure. As any other product with an open licence (Apache Software Licence), Apache's vulnerabilities are always detected in small periods of times and patches are released. However, if the apache web server is not constantly updated, the probabilities for the system to be compromised increase. In an environment in which Apache web server is the most popular web server around, vulnerability in this software could be catastrophic.

6.3.5 Implications

As demonstrated in the past examples, application layer attacks will always be part of computer system. This is even worse in systems in which the underlying OS does not protect against these attacks. If the attacker is able to compromise the application, the attacker gets all the privileges that the compromised application has, that is, the privileges of the user. Consequently, the attacker is able to access all the resources to which the user has access and even more, all these resources are at the discretion of the attacker. Also, in a DAC OS, the attacker can escalate privileges and get system administrator privileges to compromise the entire system.

6.4 Software Vulnerabilities and Malware

Hackers have always been interested in the development of software which is able to take advantage of vulnerabilities in software applications to infiltrate computer systems. With the widespread use of the Internet, attackers could use similar vulnerabilities in interconnected systems and spread malware in thousands of computers in less than a minute. Viruses, worms and Trojans are common examples of malware that has been developed with the unique purpose to exploit software vulnerabilities.

In order to mitigate the risk from malware, organisations have to reduce the vulnerabilities in their systems. Computer systems have to be protected using tools like antivirus and keep up-to-date with the last releases of malware fingerprints. Also, it is important to keep the software applications patched with the last security updates. However, malware could be released even before a new update for the antivirus or a new patch is released. Consequently, mechanisms from the OS have to be in place in order to minimise the damage from these attacks.

In this section some examples of malware are going to be described in order to understand the attacks to which applications are targeted. Also, malware needs to be understood in order to mitigate its damage and be prepared for its inevitable existence. The damage caused by the malware introduced in this section can be mitigated or eradicated with the use of SELinux Profiles

6.4.1 Backdoors

A backdoor is not malware developed by a hacker which exploits vulnerabilities in a software application, but could be as disastrous as any malware. A backdoor could be defined as a “hole” in a system or application software deliberately left in order to bypass normal authentication, remotely access a computer system, obtaining access to restricted resources and so on. A backdoor could be left by the designer or maintainers of the system or applications to gain access on specific situations. However, a backdoor could also be created by an attacker. An attacker could insert or attach malicious code to legitimate application software of a system allowing the attacker to easily bypass security mechanisms.

An attacker could create a backdoor without having to modify the application code. The attacker can modify the compiled code in order to make it recognise code that will call the backdoor. Consequently, when the user runs the application, it will call the backdoor if certain parameters are provided. Viruses, worms and Rootkits install backdoors in the compromised system so that attackers can bypass security mechanisms for current or future attacks.

6.4.2 Viruses, Worms and Trojans

Trojans are malicious software programs which appear to be legitimate programs which perform certain actions but they add a hidden subversive functionality. The use of network systems to share information has demonstrated to be a perfect means to spread Trojans. Computer systems are infected with Trojans when users receive by email or download software from the Internet and the software is executed in the system. Once the Trojan is executed in the victim’s system then Trojans may exploit a buffer overflow, create a backdoor, exploit race conditions, or perform brute force attacks. Trojans may have different purposes but the basic one is exploit application vulnerabilities. In these days, the most common exploit performed by a Trojan is the installation of backdoor programs.

An example of a Trojan is the *Bandook Rat*. The *Bandook Rat* is a Remote Access Trojan which infects MS Windows NT family systems such as MS Windows XP and MS Vista. This Trojan creates a backdoor program which allows the attacker to remotely control a remote computer system. Plenty of other Trojans are out there freely available to hackers so that they can exploit vulnerabilities in software applications and “own” computer systems.

A virus is a software program which attaches itself to a program or file and its main purpose is to make copies of itself and spread those copies into other computers without permission or knowledge of the user. Trojans are not Viruses in the way that they do not propagate by self-replications. However, Viruses and Worms could be considered a form of Trojan horses in the way that they exploit vulnerabilities in a computer system while masquerading as legitimate programs. Viruses need help from humans to spread themselves into another computer by sharing files of sending e-mails with viruses attached to these files.

A worm is a virus which has the same purpose of a virus but it makes use of networks to spread its copies without help from humans. Because worms do not need help from humans to spread itself through the network, they can send hundreds or thousands of copies of itself from an infected computer. These could cause a considerable increase in memory and network bandwidth overhead which could lead Web Servers, network servers and individual computers to stop responding.

One of the most common examples of worms is the Blaster and the SQL Slammer Worms. The Blaster worm was created to take advantage of buffer overflow vulnerabilities in the MS OS, specifically MS Windows NT family such as MS Windows NT, MS 2000 and MS Windows XP. The Blaster Worm is a DoS Attack Trojan whose main purpose was to issue a DoS attack against the Windows Update site. SQL Slammer exploited a buffer overflow vulnerability in the MS SQL Server DB. This worm was so effective in its attack that was able to infect 75,000 hosts (90% in the first 10 minutes of its release), caused outages in approximately 13,000 Bank of America ATMs, and even caused delays in regional flights by affecting Continental Airline's systems (Daswani et al., 2007). Even if these two types of worms were not created with destructive purposes (delete, modify, or access sensitive information) in mind, their impact was very serious. However, this gives a good idea about how worms could affect systems if they are created for malicious purposes.

6.4.3 Rootkits

Rootkits could be considered to be Trojans comprised by a set of software programs created with the unique purpose to masquerade as trustable commands in the OS so that the operations of the attacker are hidden. Once that the attacker has compromised the computer system and got root privileges (in a DAC system) an attacker would like to leave something so that it would be easier to "own" the computer system again. Compromising a computer system could be a very complicated task and even more getting root privileges. Once this is done, the attacker could install a Rootkit in the system so that the attacker keeps the privileges obtained. Rootkits could use a variety of techniques to masquerade making it think that the system is secure.

When organisations are aware of the security risks existent while using the Internet and other public networks, they make use of security systems, networks and end-user equipment to prevent malware from getting into the computer systems. However, most of these organisations are not prepared for rootkits for which the main purpose is to remain hidden from the system administrator and any existent anti-virus. A company which manages sensitive information could be compromised without being aware of it while the rootkit sends information and hides its activities. The proliferation and sophistication of rootkits in these days is alarming. Rootkits are able to hide not only any malware existent in the computer system, but also the information that is being obtained and sent through the network. Nevertheless, the proliferation of rootkits is due to the ease with which they can be implemented. Unpatched security holes in OS and application software (mainly in MS Windows systems) increases the likelihood of rootkits to be installed.

6.5 SELinux Profiles and Sandboxes

SELinux is able to provide the flexibility and granularity required to protect sensitive healthcare information through the use of SELinux Profiles. As specified in Chapter 5, SELinux Profiles are used to restrict the environment in which processes running on behalf of the user are allowed to operate. The user is assigned to a SELinux UID which determines the privileges authorised for the user. The SELinux Profile determines the roles, domains and types that the user is able to access.

The roles authorised to the user are going to determine the set of domains to which the user is authorised to access. These domains are restricted to specific access permissions over objects. In this way the user is restricted to access only those resources authorised to the domains which his/her roles are authorised to access. This is the way in which sandboxes are created and assigned to users with SELinux Profiles. For example, let us assume that the *Diagnostic Application* process running on behalf of a doctor is labelled with the domain *hc_diag_doc_t* and is allowed to access only files with domain *hc_diag_file_t*. Also, a user *Bob* is authorised to access the role of the doctors which is authorised to access the domain *hc_diag_doc_t*. Once the user *Bob* executes the *Diagnostic Application* and consequently enters to the domain *hc_diag_doc_t*, *Bob* is going to be restricted to only those objects and subjects authorised to the domain *hc_diag_doc_t*. In this case the sandbox is defined by the domain *hc_diag_doc_t* and there is no way in which *Bob* access objects not assigned to this sandbox (except if the user *Bob* changes to another sandbox).

In this point could be easy to confuse sandbox with domain, but these terms are not the same. Domains are types assigned to subjects in the system. Sandboxes are the authorised environment for the subject. This authorised environment consists of all the authorised access permissions that the domains has over objects and subjects in the system.

Figure 9 shows how the sandboxes can be used to restrict the damage from a compromised application. Let us assume that the domain of the service 1 (Apache) is *apache_t*. If the user is authorised to access the domain *apache_t*, the user is restricted only to those permissions authorised for the domain *apache_t*. If the application is compromised, the user still can access only those objects and subjects authorised to the domain *apache_t* and no more.

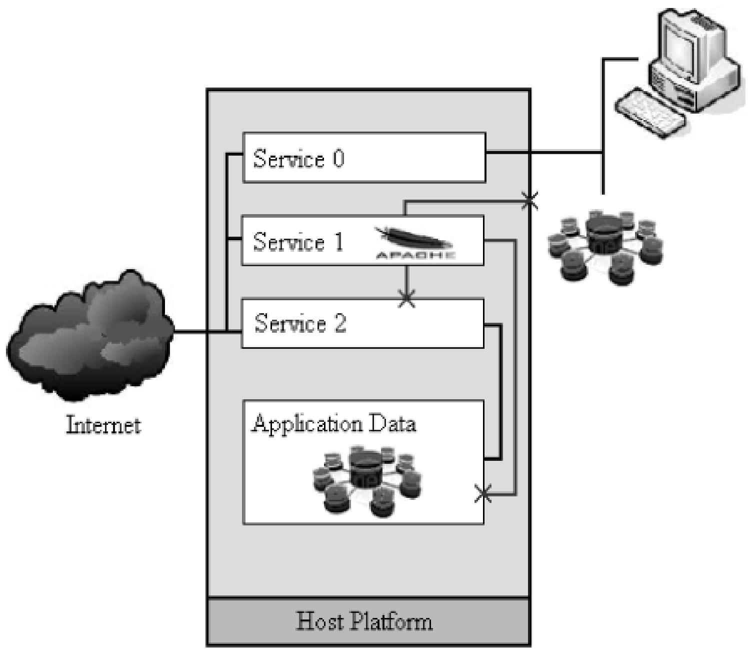


Figure 8. Sandboxes with SELinux

6.6 Healthcare Attack Scenario

The attack scenario explained in this section is based on the *Diagnostic Application* introduced in Chapter 5. 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 *Diagnostic Application*.

In the normal *Diagnostic* Application, only the physicians, nurses and patients are authorised to access the files in the patients’ directories. These files contain diagnostic information that can be related to a specific patient. Researchers are authorised only to access files in the researchers’ directory. However, the backdoor created by the attacker allows a researcher to have access to files in patients’ directories, compromising sensitive information of the patients. The *Diagnostic Application* was modified so that when the attacker specifies the *backdoor* parameter, the researcher is able to access sensitive patients’ information.

The following section describes how this attack would compromise a system which implements DAC mechanisms at the OS layer. This is follows by an explanation about how MAC mechanisms at the OS layer can contain this attack.

6.6.1 DAC Context

Security attributes for subjects in traditional Unix-like systems are the user and group IDs of the user that accessed the subjects. The objects security attributes are the permission bits and the file user and group IDs which are designated by the system administrator or during the creation of the file.

The *Diagnostic Application* is coded in such a way that the only way to create and check diagnostic reports of patients is through this application. Users should not be able to access the files created by the *Diagnostic Application* even if they are in the same computer. In a DAC system in order to limit access to all other users except to the *Diagnostic Application*, the *Diagnostic Application* has to be run with root privileges. In this way, only the *Diagnostic Application* can access the patient and researchers directories which are at system administrator level.

Table 8 shows the users with their respective DAC security attributes which were created as part of the prototype of the research.

User Type	Linux UID	Linux GID
Doctor	Drpaul (501)	healthcare (51001)
Researcher	resjohn (502)	healthcare (51001)

Table 8. Users' DAC Security Attributes

Remember that user and group IDs in Unix-like systems are managed internally as integers, but for simplicity we use the string representation in this thesis.

Table 9 shows the directories created to store the *Diagnostic Application* files along with their DAC security attributes.

File name	Path	Permission bits	Owner UID	GID
diagnosis.di	/healthcare/db/patients/<patient>	rw- --- ---	root	healthcare
diagnosis.di	/healthcare/db/researchers	rw- --- ---	root	healthcare

Table 9. Files and Directories DAC Secure Attributes

The two *diagnosis.di* files are created with the content that corresponds to each user, that is, for the patients it will contain the complete diagnostic record while for researchers only the information required for statistical purposes. Both files are created with read and write permission bits activated only for the owner of the information which is the root user. In this case only those with root privileges are able to read or write these files making it impossible for a normal user to disclose the content of the files.

Table 10 shows the permission bits assigned to the *Diagnostic Application* executable.

Executable file	Permission bits	Owner ID	Group ID
/bin/diag_sys	rws r-x ---	root	healthcare

Table 10. Executable File DAC Security Attributes

Note that the *setuid* permission bit is used within the permission bits of the *diag_sys* executable file. The *setuid* permission bit allows those processes with the permission to execute the file to change its effective UID (which is the one of the user running the application) to that of the file owner. That is, in this case processes which are run by the users *drpaul* and *resjohn* will change their UID to that of the root user and thus acquiring root privileges. The use of the *setuid* permission bit is commonly used in critical application such as the *passwd* program which has direct access to the *shadow* file containing the encrypted passwords of all users in the system.

When the users *drpaul* or *resjohn* log in, they are authorised to access neither the directories nor the files containing sensitive information. In this way the information remains secure against any attempt to intentionally or unintentionally disclose this information. Both users are allowed to access this information only through the use of the *Diagnostic Application*. This is done by executing the file *diag_sys*.

However, if the *diag_sys* binary file is modified and a backdoor is attached, then the attacker would not only be able to access patient's sensitive information, but also to every resource in the system. This is because those who execute the *diag_sys* file are granted with root privileges. This is a clear violation to the principle of least privilege since those who run the *diag_sys* program are able to access all the resources in the system. This is one of the main weaknesses in standard Unix-like systems and in any other DAC system in which applications running as root have full access to the resources in the system.

In this example, the user *resjohn* is able to specify the parameter *backdoor* while executing the *diag_sys* program and access the diagnostic records of any patient. Because the *diag_sys* program is run with the privileges of the root user, this access is authorised. Once the user *resjohn* compromises the file, there are no limits on the accesses that the user can do.

6.6.2 MAC 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.

In order to exemplify the way in which SELinux restricts the *Diagnostic Application* let us simplify the DAC attributes specified in the previous section and leave all access restrictions to SELinux.

The following users and respective secure attributes are created:

Linux UID	Linux GID	SELinux (UID)	Role	Shell Process Security Context
drpaul (501)	healthcare (51001)	hc_doc_u	hc_doc_r	hc_doc_u:hc_doc_r:hc_doc_t
resjohn (502)	healthcare (51001)	hc_res_u	hc_res_r	hc_res_u:hc_res_r:hc_res_t

Table 11. Seucity Attributes for the Diagnostic Application

As shown in Figure 4, Chapter 5, once user *paul* is appropriately authenticated, the user is assigned to his Linux UID *drpaul* and mapped to his SELinux UID *hc_doc_u*. The SELinux UID determines the SELinux Profile of the doctor, which in this case authorise the SELinux UID *hc_doc_u* to access the role *hc_doc_r*. Once the user starts the shell process, the shell process is assigned with the security context: *hc_doc_u:hc_doc_r:hc_doc_t*. In a similar way the shell process running on behalf of the user john (researcher) is assigned with the security context: *hc_doc_u:hc_doc_r:hc_doc_t*.

The files created by the diagnostic application are stored as follows:

File name	Path	Permission bits	Owner UID	Linux GID	Files Security contexts
diagnosis.di	/healthcare/db/patients/<patient_name>	rw- rw- ---	root	healthcare (51001)	hc_doc_u:object_r: hc_pnt_dbfile_di_t
diagnosis.di	/healthcare/db/researchers	rw- rw- ---	root	healthcare (51001)	hc_doc_u:ch_object_r: hc_res_dbfile_di_t

Table 12. Security Attributes for files of the Diagnostic Application

The D*iagnostic Application* stores the diagnostic reports in two files with the same name *diagnosis.di*. Each file contains specific information for physicians and researchers. Both files are assigned read and write permission for the owner and the group, so that all those which belong to the healthcare group can read and write diagnostic files. However, this does not mean that the user *resjohn* can access these files. In a DAC OS this would be possible, but in SELinux access is granted according to the specification on the SELinux Policy and do not rely only on DAC access control checks. In a similar manner, every directory in the path of the *diagnosis.di* files is create with the “rwx rwx ---” permission bits activated in order to allow anyone in the group *healthcare* to read, write and access in the directories. But again, access to these directories will depend on the authorised access permissions specified in the SELinux Policy.

Because of the high granularity while using TE, both files are labelled with different types which are unique to the file. The types are: *hc_pnt_dbfile_di_t* for the patients’ file and *hc_res_dbfile_di_t* for the researchers file. When the physicians create a diagnostic report, these files are created by the *Diagnostic Application* and are assigned with the SELinux UID of the physician. That is, patients and researchers file will have a similar SELinux UID which is *hc_doc_u*. However, this does not mean that the physician owns the files; access to the resources for the physicians still depends on the SELinux Policy.

The diagnostic application executable file is created with the following permissions:

Executable File	Permission bits	Owner ID	Group ID	Executable File Security Context
/bin/diag_sys	rwx rwx ---	root	healthcare (51001)	system_u:object_r: hc_diag_sys_exec_t

Table 13. Security Attributes for the Diagnostic Application Executable File

Even if the *diag_sys* executable file has the read, write and execute 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 authorised to execute the file in the SELinux Policy. Also, the appropriate transition rules would have to be specified in order to authorise the user to access the domains of the *Diagnostic Application*. For example, the TE rules needed to authorise the domain transition of the shell process domain *hc_doc_t* to the *Diagnostic Application* process domain *hc_doc_diag_t* are as follows:

```
type_transition hc_doc_t hc_diag_sys_exec_t : file hc_doc_diag_t

allow hc_doc_t hc_diag_sys_exec_t : file { execute }

allow hc_doc_diag_t hc_diag_sys_exec_t : file entrypoint;

allow hc_doc_t hc_doc_diag_t : process { transition };
```

These AVR will authorise the shell process running on behalf of the user *drpaul* to transit into the *hc_doc_diag_*t corresponding to the *Diagnostic Application* process running on behalf of a physician. The domain type *hc_doc_diag_t* could be considered to be the sandbox which is going to restrict the operations of the *Diagnostic Application* running in behalf of any physicians.

In a similar way, the researchers' shell processes have to be authorised to transit into the domain of the *Diagnostic Application* process running on behalf of researchers. In this case, instead of transiting into the domain *hc_doc_diag_t* the researchers are authorised by the SELinux Profile to access the domain *hc_res_diag_t*. The *hc_res_diag_t* domain represents the sandbox that restricts the operations of the *Diagnostic Application* running on behalf of any researcher.

The authorised types, domains and access permissions defined by the SELinux Profiles are shown in Table 14. For example, the domain *hc_doc_t* belonging to a shell process running on behalf of a physician, is able to transit to the domain *hc_doc_diag_t* and create files with the type *hc_pnt_dbdir_usr_t.*

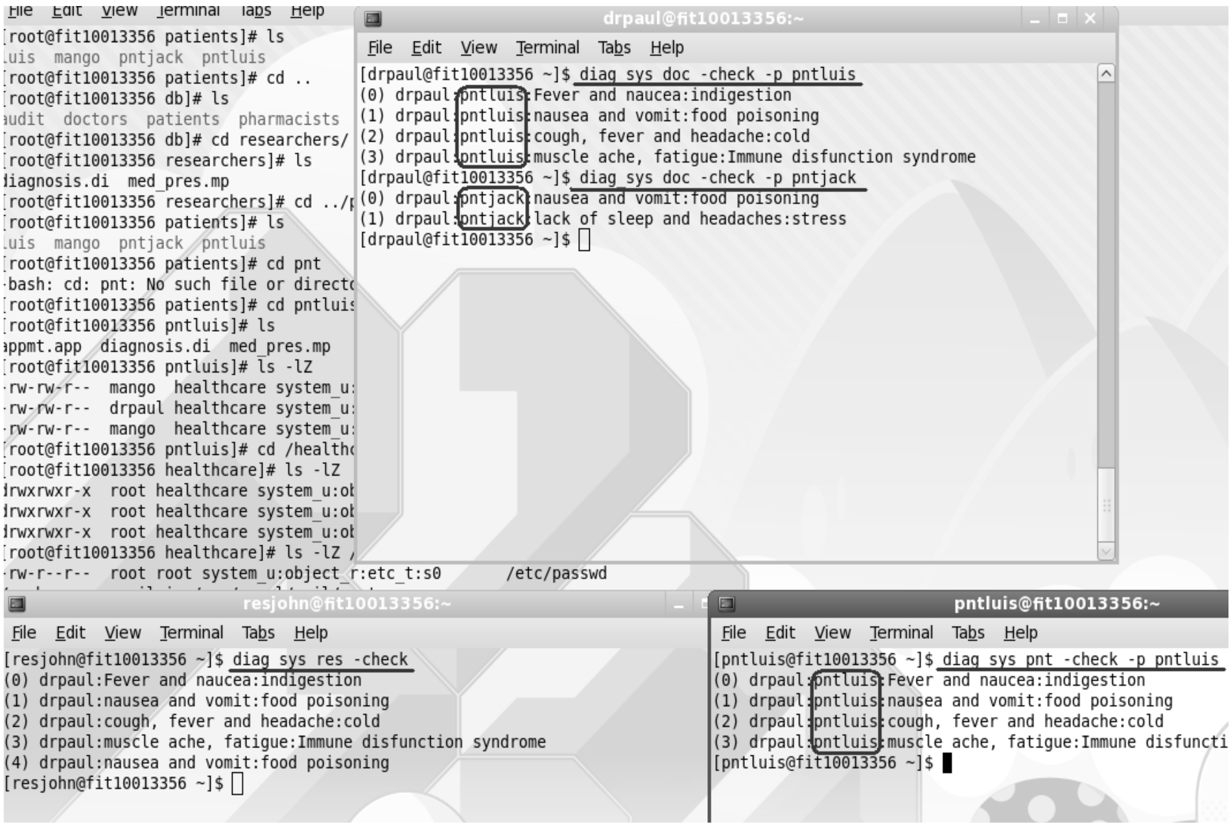
Shell Process domains	**Diagnostic Application domains**	**File Types**	**Object Classes**	**Permissions**
hc_doc_t	hc_doc_diag_t	hc_topdir_t	Dir	Search_dir_perms
		hc_topdir_db_t	Dir	Search_dir_perms

		hc_pnt_dbdir_t	Dir	add_entry_dir_perms create_dir_perms search_dir_perms
		hc_pnt_dbdir_usr_t	Dir	add_entry_dir_perms create_dir_perms search_dir_perms
		hc_pnt_dbfile_di_t	File	create_file_perms rw_file_perms read_file_perms
		hc_res_dbdir_t	Dir	create_dir_perms add_entry_dir_perms
		hc_res_dbfile_di_t	File	create_file_perms rw_file_perms
hc_res_t	hc_res_diag_t	hc_topdir_t	Dir	Search_dir_perms
		hc_topdir_db_t	Dir	Search_dir_perms
		hc_res_dbdir_t	Dir	Search_dir_perms
		hc_res_dbfile_di_t	File	read_file_perms

Table 14. SELinux Profiles for Physicians and Researchers.

In Table 14 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 *Diagnostic Application* is executed by a researcher, the shell process with domain type *hc_res_t* will transit into the domain *hc_res_diag_t*. This domain is only authorised to read files with type *hc_res_dbfile_di_t* and access directories with type *hc_res_dbdir_t*. If the researchers try to access a directory or file with types not specified in this table, the access is denied.

As shown in Figure 10, doctors and patients are allowed to read files in the patients' directories containing diagnostic report information that can be related to a specific patient. On the other hand, researchers can only access files in the researchers' directory containing information which cannot be related any particular patient. In this example the doctor *drpaul* check the diagnostic records for patients *pntluis* and *pntjack* which are displayed with the names of the patients and their symptoms. The researcher *resjohn* can check the diagnostic reports, but these are displayed without the patients’ name, thus the information cannot be related with any patient in particular.



```
[root@fit10013356 patients]# ls
uis  mango  pntjack  pntluis
[root@fit10013356 patients]# cd ..
[root@fit10013356 db]# ls
udit  doctors  patients  pharmacists
[root@fit10013356 db]# cd researchers/
[root@fit10013356 researchers]# ls
iagnosis.di  med_pres.mp
[root@fit10013356 researchers]# cd ../p
[root@fit10013356 patients]# ls
uis  mango  pntjack  pntluis
[root@fit10013356 patients]# cd pnt
-bash: cd: pnt: No such file or directo
[root@fit10013356 patients]# cd pntluis
[root@fit10013356 pntluis]# ls
ppmt.app  diagnosis.di  med_pres.mp
[root@fit10013356 pntluis]# ls -lZ
-rw-rw-r--  mango  healthcare system_u:
-rw-rw-r--  drpaul healthcare system_u:
-rw-rw-r--  mango  healthcare system_u:
[root@fit10013356 pntluis]# cd /health
[root@fit10013356 healthcare]# ls -lZ
drwxrwxr-x  root healthcare system_u:ob
drwxrwxr-x  root healthcare system_u:ob
drwxrwxr-x  root healthcare system_u:ob
[root@fit10013356 healthcare]# ls -lZ /
-rw-r--r--  root root system_u:object_r:etc_t:s0       /etc/passwd
```

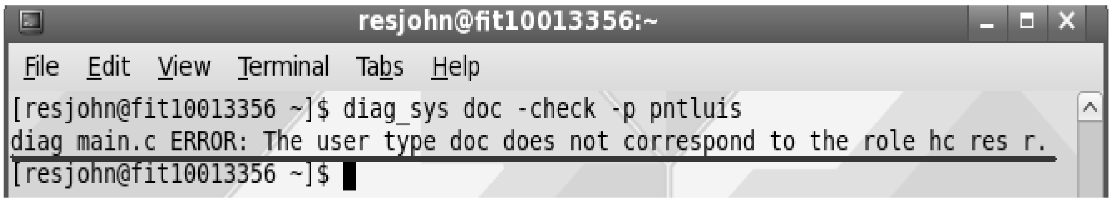
```
[drpaul@fit10013356 ~]$ diag sys doc -check -p pntluis
(0) drpaul:pntluis:Fever and naucea:indigestion
(1) drpaul:pntluis:nausea and vomit:food poisoning
(2) drpaul:pntluis:cough, fever and headache:cold
(3) drpaul:pntluis:muscle ache, fatigue:Immune disfunction syndrome
[drpaul@fit10013356 ~]$ diag_sys doc -check -p pntjack
(0) drpaul:pntjack:nausea and vomit:food poisoning
(1) drpaul:pntjack:lack of sleep and headaches:stress
[drpaul@fit10013356 ~]$ 
```

```
[resjohn@fit10013356 ~]$ diag sys res -check
(0) drpaul:Fever and naucea:indigestion
(1) drpaul:nausea and vomit:food poisoning
(2) drpaul:cough, fever and headache:cold
(3) drpaul:muscle ache, fatigue:Immune disfunction syndrome
(4) drpaul:nausea and vomit:food poisoning
[resjohn@fit10013356 ~]$ 
```

```
[pntluis@fit10013356 ~]$ diag sys pnt -check -p pntluis
(0) drpaul:pntluis:Fever and naucea:indigestion
(1) drpaul:pntluis:nausea and vomit:food poisoning
(2) drpaul:pntluis:cough, fever and headache:cold
(3) drpaul:pntluis:muscle ache, fatigue:Immune disfuncti
[pntluis@fit10013356 ~]$ 
```

Figure 9. Diagnostic System Normal Behaviour

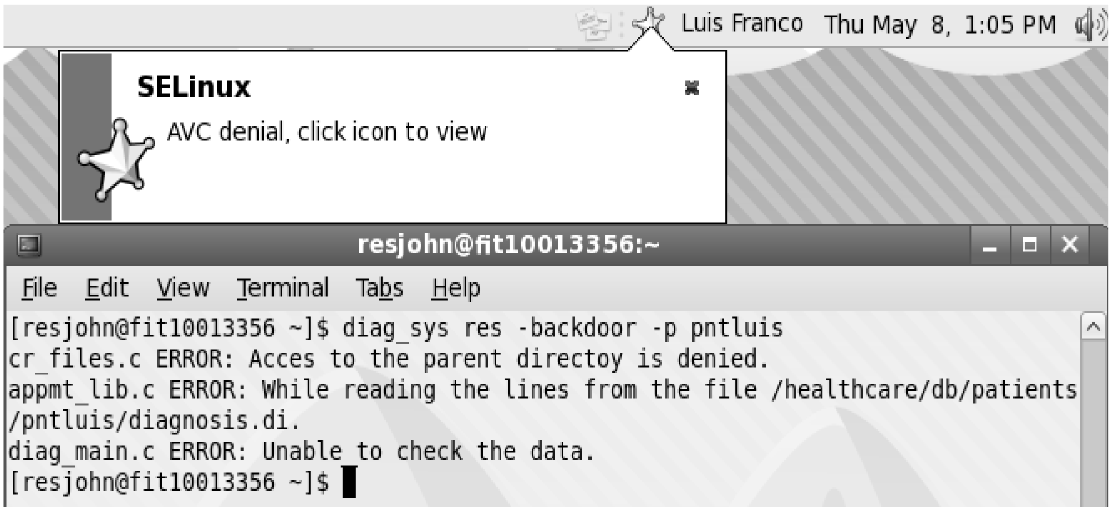
If the researcher tries to access the medical record of a particular patient or trick the application masquerading as a physician, the diagnostic systems displays the message shown in Figure 11.



```
[resjohn@fit10013356 ~]$ diag_sys doc -check -p pntluis
diag main.c ERROR: The user type doc does not correspond to the role hc res r.
[resjohn@fit10013356 ~]$ 
```

Figure 10. Diagnostic System Error Message

Let us assume that the *Diagnostic Application* was modified and a backdoor was attached to the application in order to allow a researcher to specify the parameter *backdoor* and access patients’ sensitive information. The *Diagnostic Application* while executed by a researcher runs within the boundaries of the *hc_res_diag_sys_t* domain. This domain is authorised to access only files with the type *hc_res_dbfile_di_t*. Consequently, if the *Diagnostic Application* process on behalf of a researcher tries to access any file without the type *hc_res_dbfile_di_t*, SELinux denies the access and creates an AVC message in the audit log as shown in Figure 12.

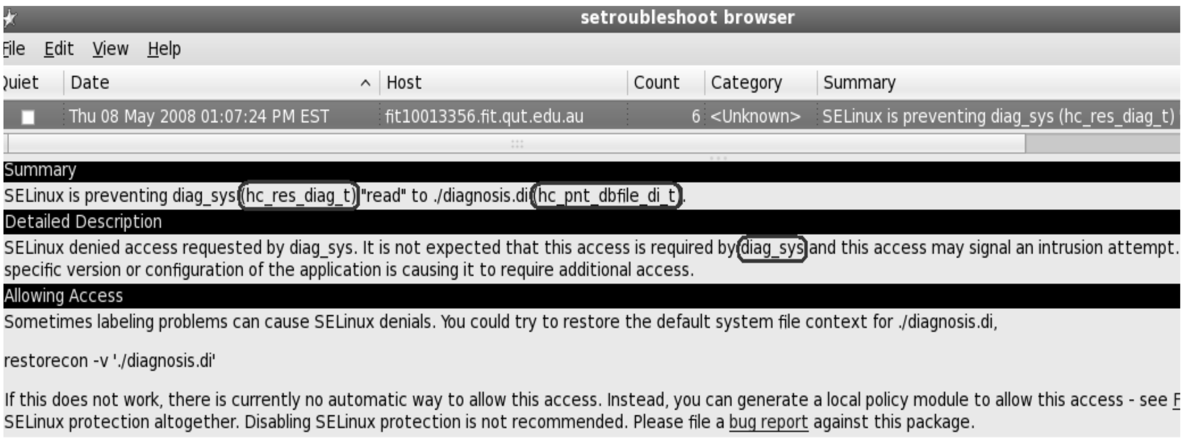

Luis Franco Thu May 8, 1:05 PM
resjohn@fit10013356:~
File Edit View Terminal Tabs Help

SELinux
AVC denial, click icon to view

```
[resjohn@fit10013356 ~]$ diag_sys res -backdoor -p pntluis
cr_files.c ERROR: Acces to the parent directoy is denied.
appmt_lib.c ERROR: While reading the lines from the file /healthcare/db/patients
/pntluis/diagnosis.di.
diag_main.c ERROR: Unable to check the data.
[resjohn@fit10013356 ~]$
```

Figure 11. Diagnostic System's Backdoor and AVC Message

Figure 13 shows the AVC messages as displayed in setroubleshoot. This tool helps to simplify the understanding of AVC message providing an understandable and user-friendly description of the denied access and its possible solution.



uiet	Date	Host	Count	Category	Summary
■	Thu 08 May 2008 01:07:24 PM EST	fit10013356.fit.qut.edu.au	6	<Unknown>	SELinux is preventing diag_sys (hc_res_diag_t)

Summary

SELinux is preventing diag_sys (hc_res_diag_t) "read" to ./diagnosis.di (hc_pnt_dbfile_di_t).

Detailed Description

SELinux denied access requested by diag_sys. It is not expected that this access is required by diag_sys and this access may signal an intrusion attempt.
specific version or configuration of the application is causing it to require additional access.

Allowing Access

Sometimes labeling problems can cause SELinux denials. You could try to restore the default system file context for ./diagnosis.di,

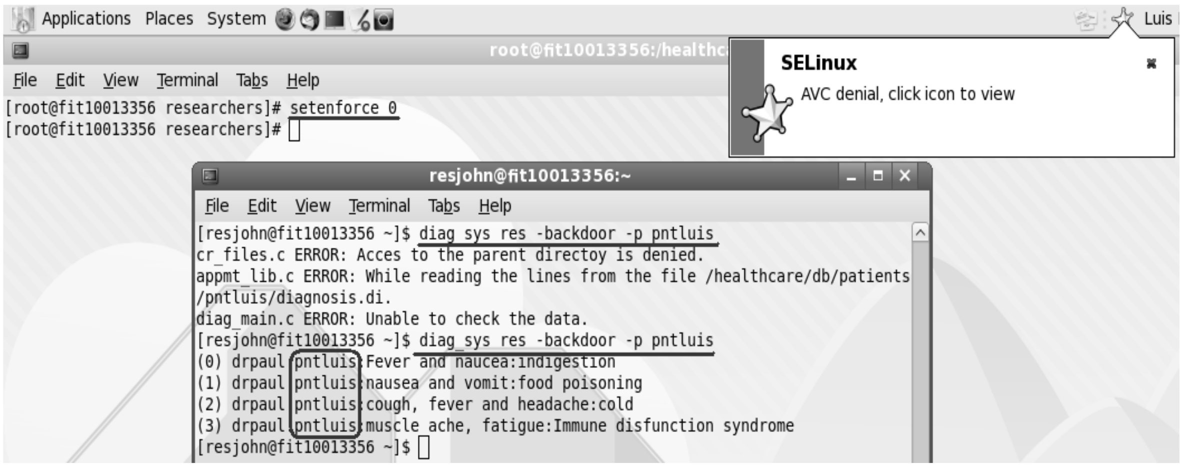
restorecon -v './diagnosis.di'

If this does not work, there is currently no automatic way to allow this access. Instead, you can generate a local policy module to allow this access - see F
SELinux protection altogether. Disabling SELinux protection is not recommended. Please file a bug report against this package.

Figure 12. setroubleshoot browser - backdoor denial

The AVC message shown in Figure 13 specifies that the process with domain *hc_res_diag_t* is trying to access an unauthorised file. The message specifies that the *Diagnostic Application* process with domain hc*_res_diag_t*.is trying to access the *./diagnosis.di* file with type *hc_pnt_dbfile_di_t.*

In order to demonstrate what would happen in case that SELinux is not used. Figure 14 shows the results of the domain *hc_res_diag_*t accessing a file with type *hc_pnt_dbfil_di_*t when SELinux is set in permissive mode. That is, the kernel does not enforce the SELinux policy any more, but the AVC Messages are still created in the audit log. Note that the command *setenforce* is used to disable SELinux.


Applications Places System
Luis
root@fit10013356:/healthc
File Edit View Terminal Tabs Help
resjohn@fit10013356:~
File Edit View Terminal Tabs Help

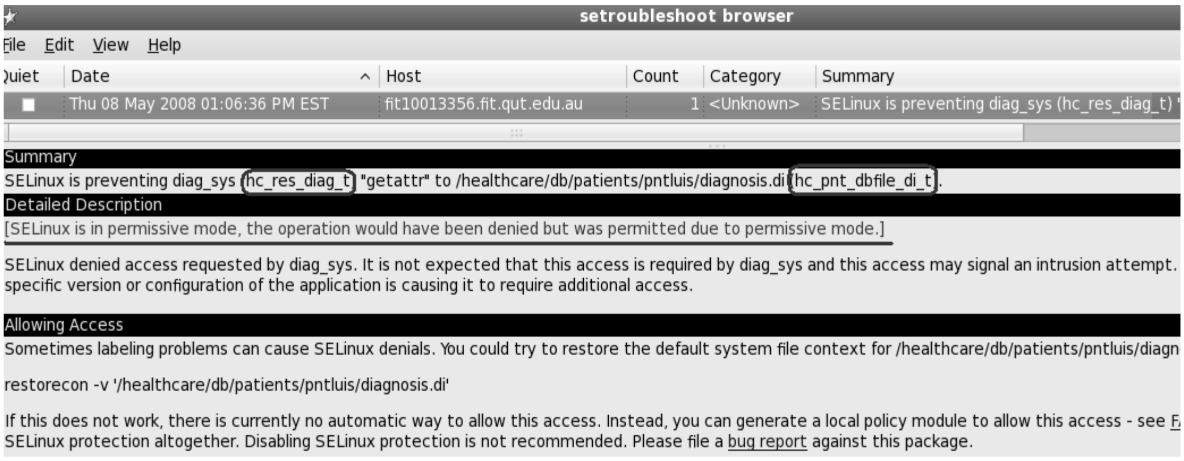
```
[root@fit10013356 researchers]# setenforce 0
[root@fit10013356 researchers]#
```

SELinux
AVC denial, click icon to view

```
[resjohn@fit10013356 ~]$ diag_sys_res -backdoor -p pntluis
cr_files.c ERROR: Acces to the parent directoy is denied.
appmt_lib.c ERROR: While reading the lines from the file /healthcare/db/patients
/pntluis/diagnosis.di.
diag_main.c ERROR: Unable to check the data.
[resjohn@fit10013356 ~]$ diag_sys_res -backdoor -p pntluis
(0) drpaul:pntluis:Fever and naucea:indigestion
(1) drpaul:pntluis:nausea and vomit:food poisoning
(2) drpaul:pntluis:cough, fever and headache:cold
(3) drpaul:pntluis:muscle ache, fatigue:Immune disfunction syndrome
[resjohn@fit10013356 ~]$
```

Figure 13. Diagnostic System's Backdoor - SELinux Permissive mode

As shown in Figure 14, once SELinux is set in permissive mode, the researcher is able to use the backdoor attached in the *Diagnostic Application* and access files in the patients directories. However, SELinux still creates an AVC message in the audit log that can be seen in setrobleshoot and is shown in Figure 14.



Quiet	Date	Host	Count	Category	Summary
☐	Thu 08 May 2008 01:06:36 PM EST	fit10013356.fit.qut.edu.au	1	<Unknown>	SELinux is preventing diag_sys (hc_res_diag_t) "

Summary
SELinux is preventing diag_sys (hc_res_diag_t) "getattr" to /healthcare/db/patients/pntluis/diagnosis.di (hc_pnt_dbfile_di_t).

Detailed Description
[SELinux is in permissive mode, the operation would have been denied but was permitted due to permissive mode.]

SELinux denied access requested by diag_sys. It is not expected that this access is required by diag_sys and this access may signal an intrusion attempt.
specific version or configuration of the application is causing it to require additional access.

Allowing Access
Sometimes labeling problems can cause SELinux denials. You could try to restore the default system file context for /healthcare/db/patients/pntluis/diagn

restorecon -v '/healthcare/db/patients/pntluis/diagnosis.di'

If this does not work, there is currently no automatic way to allow this access. Instead, you can generate a local policy module to allow this access - see F
SELinux protection altogether. Disabling SELinux protection is not recommended. Please file a bug report against this package.

Figure 14. AVC Message with SELinux in Permissive Mode

The AVC message shown in Figure 14 specifies that the domain hc_res_diag_t is accessing the file diagnosis.di with type hc_pnt_dbfile_di_t. It also specifies that this permission is allowed because "SELinux is in permissive mode", but if this would no be the case, the access would be denied.

6.7 Conclusion

This chapter demonstrated that security solutions at the application layer alone are not enough to satisfy security requirements in HIS. Computer systems have to be constructed with the support from the underlying OS. Viruses, Worms and Trojans are attacks commonly used to compromise applications and to access unauthorised resources in the system in which the applications are running. In OSs that implement DAC mechanisms, the damage from compromised applications cannot be contained. SELinux is a preferred solution in order to minimise 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. This behaviour was demonstrated following the *Diagnostic Application* scenario used in the prototype of this research. The scenario demonstrated that SELinux can effectively contain the damage from compromised applications. This was achieved through the creation of sandboxes for the *Diagnostic Application* while running on behalf of specific users.

Chapter 7

Conclusions

Contents:

7 Conclusions

7.1 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 organisations 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 unauthorised accesses, access control mechanisms have to be implemented at the OS layer. Currently, most systems are constructed using OSs which implement 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 authorised 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 organisations.

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 organisations 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 provide MAC at the OS layer were based on MLS. OSs that implement 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 organisations 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 authorised 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 behaviour and try to access unauthorised resources, SELinux denies the access since it is outside of the boundaries of the sandbox, that is, out of the authorised permissions for the domain.

RBAC in SELinux has been recognised 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 authorised 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 authorised 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 *Diagnostic Application* 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 minimise 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 implement 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 minimise 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 behaviour was demonstrated describing an attack scenario using the *Diagnostic 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 *Diagnostic Application* while running on behalf of specific users.

7.2 Future Work

Because of the time restriction of this research, the use of SELinux was limited to protecting healthcare information only at the OS layer. With the development of the security server manager project by Tresys Technologies, it is possible to extend the granularity and flexibility of SELinux to the application layer.

Also, in this research RBAC was used based on RBAC0. SELinux also allows the implementation of role hierarchies. In future research, the use of role hierarchies can simplify the policy configuration. As stated by Henricksen et al. (2007), role hierarchies in HIS are very useful features.

Finally, the proposed solution in this research is just a small part in the protection of HIS. HIS have to be protected following a holistic approach in which security mechanisms have to be provided in four layers of protection: application, OS, network and hardware. This research has to be integrated with the OTHIS architecture proposed by Liu et al. (2008a). In the OTHIS architecture, SELinux form part of the Health Information Access Control (HIAC) domain. SELinux have to interact with the Health Information Application Security (HIAS) and the Health Information Network Security (HINS) domains.

References

8 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. Anderson, R. (2001). *Security Engineering: A Guide to Building Dependable Distributed Systems*. Hoboken, N.J.: John Wiley & Sons.

3. *Apache Tomcat* (n.d.). Retrieved May 30, 2008, from

   http://tomcat.apache.org/security-5.html

4. Arnott, S. (2004). Smartcard ID for NHS staff. *Computing*. Retrieved 05 May, 2008, from

   http://www.computing.co.uk/computing/news/2070661/smartcard-id-nhs-staff.

5. *Assembly Bill No. 1298* (2007). Retrieved April 15, 2008, from

   http://www.n-tegritysolutions.com/ab_1298_bill_20071014_chaptered.pdf

6. Bacon, J., Moody, K., & Yao, W. (2003). Access Control and Trust in the Use of Widely Distributed Services. *Software: Practice and Experience*, 33(4), 375–394.

7. Badger, L., Sterne, D. F., Sherman, D. L., Walker, K. M., and Haghighat, S. A. (1995). Practical domain and type enforcement for UNIX. *Proceedings of the 1995 IEEE Symposium on Security and Privacy*. Oakland, CA, USA.

8. Bauer, M. (2007). Paranoid Penguin: Introduction to SELinux. *Linux Journal*, 2007(155), 13.

9. BBC News (2008). *Privacy fear over NHS card loss*. Retrieved May 24, 2008, from

   http://news.bbc.co.uk/2/hi/health/7230512.stm

10. Bennett, K., Rigby, M., and Budgen, D. (2006). Role based access control – a solution with its own challenges. *IEE Proceedings – Software*, 153(1), 1-3.

11. Blobel, B. (2007). Comparing approaches for advanced e-health security infrastructures. *International Journal of Medical Informatics*, 76(5-6), 454-459.

12. Brevetti, F. (2006). Fingerprint sensors improve security of PCs. *Oakland Tribune*. Retrieved 05 May, 2008, from

    http://findarticles.com/p/articles/mi_qn4176/is_20060814/ai_n16648559

13. Clark, D., and Wilson, D. (1987). A comparison of commercial and military computer security policies. *Proceedings of the 1987 IEEE Symposium on Security and Privacy*. Oakland, CA, USA.

14. CNN (2008). *Google ventures into health records*. Retrieved April 16, 2008, from

http://www.cnn.com/2008/TECH/02/21/google.records.ap/

15. Cowan, C. (2001, April 11). *Linux Security Module Interface*. Retrieved April 3, 2008, from

http://marc.info/?l=linux-kernel&m=98695004126478&w=2

16. Dalton, C., and Choo, T. (2001). An Operating System Approach to Securing e-Services. *Communications of the ACM*, 44(2), 58-64.

17. Daswani, N., Kern, C., and Kesavan, A. (2007). *Foundations of Security: What Every Programmer Needs to Know*. Berkeley, CA: Apress.

18. Department of Defence. (1985). *Trusted Computer System Evaluation Criteria* (DoD 5200.28-STD Publication No. MD 20755). USA National Computer Security Centre.

19. Dixon, P. (2006). Medical Identity Theft: The Information Crime that Can Kill You. *The World Privacy Forum*. Retrieved April 20, 2008, from

http://www.worldprivacyforum.org/pdf/wpf_exsum_medidtheft2006.pdf

20. Ferraiolo, D., and Kuhn, R. (1992). Role-Based Access Control. *Proceedings of the 15th National Computer Security Conference*. Gaithersburg, Maryland, USA.

21. Ferraiolo, D., Cugini, J., and Kuhn, R. (1995). Role-Based Access Control (RBAC): Features and motivations. *Proceedings of the 11th Annual Computer Security Applications Conference*. New Orleans, LA, USA.

22. Ferraiolo, D. F., Kuhn, D. R., and Chandramouli, R. (2003). *Role-Based Access Control*. Norwood: Artech House.

23. Franco, L., Sahama, T., and Croll, P. (2008). Security Enhanced Linux to enforce Mandatory Access Control in Health Information Systems. *Proceedings of the 2nd Australasian Workshop on Health Data and Knowledge Management (HDKM 2008)*. Wollongong, NSW, Australia.

24. Goldberg, C. (2008, March 12). Heart devices vulnerable to hack attack. The Boston Globe. Retrieved March 20, 2008, from

http://www.boston.com/news/local/articles/2008/03/12/heart_devices_vulnerable_to_hack_attack/

25. Goldschmidt, P. G. (2005). HIT and MIS: implications of health information technology and medical information systems. *Communication of the ACM*, 48(10), 68-74.

26. Goldstein, D., Groen, P. J., Ponkshe, S., and Wine, M. (2007). *Medical Informatics 20/20: Quality and Electronic Health Records through Collaboration, Open Solutions, and Innovation*. Sudbury, Mass.: Jones and Bartlett Publishers.

27. Hallyn, S. (2008, February 13) *Role-based access control in SELinux*. IBM: developersWorks. Retrieved March 12, 2008, from

    http://www.ibm.com/developerworks/linux/library/l-rbac-selinux

28. Halperin, D., Heydt-Benjamin, T. S., Ransford, B., Clark, S. S., Defend, B., Morgan, W., et al. (2008). Pacemakers and Implantable Cardiac Defibrillators: Software Radio Attacks and Zero-Power Defenses. *Proceedings of the 2008 IEEE Symposium on Security and Privacy*. Oakland, California, USA

29. Hands, P., and Thomson, M. (Eds.). (1994). *American Pocket Chambers English Dictionary*. (2nd ed.). Mexico: Larousse.

30. Health Insurance Portability and Accountability Act of 1996 (n.d.). Retrieved April 10, 2008, from

    http://www.cms.hhs.gov/HIPAAGenInfo/Downloads/HIPAALaw.pdf

31. Henricksen, M., Caelli, W., and Croll, P. (2007). Securing grid data using Mandatory Access Controls. *Proceedings of the 5th Australasian Symposium on ACSW Frontiers*. Ballarat, Victoria, Australia.

32. Herzog, A. L., and Gutman, J. D. (2002). Achieving security goals with Security Enhanced Linux. *Proceedings of the IEEE Symposium on Security and Privacy*. Oakland, CA, USA.

33. Hieb, B.R, Rishel, W., Edwards, J., and Kelly, B.A. (2005). Predicts 2006: Healthcare Provider Collaboration Will Emerge, Despite Challenges. *Gartner*.

34. Hippocratic Oath (n.d.). Retrieved May 28, 2008, from

    http://info.library.unsw.edu.au/biomed/pdf/hippocraticoath.pdf

35. Hoffman, S. (2008). Microsoft Word Bug Leaves Users Open To Attack. ChannelWeb. Retrieved May 20, 2008, from

http://www.crn.com/security/206905442

36. 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.

37. Hultquist, S. (2007, April 30). Rootkits: The next big enterprise threat? *Australian Reseller News (ARN).* Retrieved March 07, 2008, from

http://www.arnnet.com.au/index.php/id;53940922;pp;1

38. Huston, T. (2001). Security Issues for Implementation of E-Medical Records. *Communications of the ACM*, 44(9), 89-94.

39. Hutchins, J. P., Caiola, A. P., Park, S., Turner, C. B., and Young, B. L. (2007). *U.S. Data Breach Notification Law: State by State*. Chicago, USA: American Bar Association.

40. INFOSEC. (n.d.). Retrieved May 29, 2008, from

http://cordis.europa.eu/infosec/src/crit.htm

41. Jean-Francois, E. (2008). Stolen laptop contains personal info of 2,500 patients. *CNN*. Retrieved April 12, 2008, from

http://edition.cnn.com/2008/US/03/25/stolen.laptop/

42. Jewell, M. (2008). Researchers Hack Defibrillators. *ABC News*. Retrieved March 20, 2008, from

http://abcnews.go.com/Technology/wireStory?id=4434207

43. Kingsbury, K. (2008, March). Medical records go digital. *TIME*. Retrieved March 24, 2008, from

http://www.time.com/time/business/article/0,8599,1723130,00.html

44. Kirk, J. (2008, January). Microsoft warns of new Excel vulnerability. *PCWorld.* Retrieved May 05, 2008, from

http://www.pcworld.com/article/id,141413-page,1/article.html

45. Krebs. B. (2006). Attacks on Unpatched IE Flaw Escalate. *WashingtonPost.* Retrieved May 26, 2008, from

http://blog.washingtonpost.com/securityfix/2006/03/attacks_on_internet_explorer_f_1.html

46. LaPadula, L. (1996). *Secure Computer Systems: Mathematical Foundations; An electronic reconstruction of the original MITRE Technical Report 2547, Volume I*. Retrieved May 20, 2008, from

    http://www.albany.edu/acc/courses/ia/classics/belllapadula1.pdf

47. Leyden, J. (2008). Storm worm botnet turns into April shower. *The Register*. Retrieved May 18, 2008, from

    http://www.theregister.co.uk/2008/05/01/storm_worm_breakup/

48. Liu, V., Caelli, W., May, L., and Croll, P. (2008a). Open Trusted Health Informatics Structure (OTHIS). *Proceedings of the 2nd Australasian Workshop on Health Data and Knowledge Management (HDKM 2008)*. Wollongong, NSW, Australia.

49. Liu, V., May, L., Caelli, W., and Croll, P. (2008b). Strengthening Legal Compliance for Privacy in Electronic Health Information Systems: A Review and Analysis. *Electronic Journal of Health Informatics*, 3(1), e3.

50. 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.

51. 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.

52. Lopez, J., Eyler, T., Galimi, J., Furlonger, D., Redshaw, P., Hieb, B. R., et al. (2006). Gartner's Top Predictions for Industry Leaders, 2007 and Beyond. *Gartner*.

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

54. Loscocco, P. A., Smalley, S. D., Muckelbauer, P.A., Taylor, R. C., Turner, S. J., and Farrel, J. F. (1998). The inevitability of failure: The flawed assumption of security in modern computing environments. *Proceedings of the 21st National Information System Security Conference*. Crystal City, Virginia, USA.

55. Lucas, H. C. (2005). *Information Technology: Strategic Decision Making for Managers*. Hoboken, N.J.: John Wiley & Sons.

56. Mayer, F., MacMillan, K., and Caplan, D. (2007). *SELinux by Example: Using Security Enhanced Linux*. Upper Saddle Rive, N.J.: Prentice Hall.

57. McLean, J. (1985). A Comment on the "Basic Security Theorem" of Bell and LaPadula. *Information Processing Letters*, 20, 67-70.

58. McVoy, L. (2003, November 5). *BK2CVS problem*. Retrieved May 6, 2008, from

http://www.ussg.iu.edu/hypermail/linux/kernel/0311.0/0635.html

59. MayoClinic. (2006) *Implantable cardioverter-defibrillators: Controlling a chaotic heart*. Retrieved March 20, 2008, from

http://www.mayoclinic.com/health/implantable-cardioverter-defibrillators/HB00003

60. McMillan, R (2007). Microsoft: Excel Vulnerable to new attack. *NetworkWorld*. Retrieved May 20, 2008, from

http://www.networkworld.com/news/2007/020507-microsoft-excel-vulnerable-to-new.html

61. McMillan, R (2006). Third Microsoft Excel attack posted. *ComputerWorld*. Retrieved May 20, 2008, from

http://www.computerworld.com/action/article.do?command=viewArticleBasic&articleId=9001349

62. McWilliams, B. (2002). Gopher Attacks Are Latest IE Security Threat. *SecurityFocus*. Retrieved May 26, 2008, from

http://www.securityfocus.com/news/464

63. Meingast, M., Roosta, T. and Sastry, S. (2006). Security and Privacy Issues with Health Care Information Technology. *Proceedings of the 28th IEEE Engineering in Medicine and Biology Society (EMBS) Annual International Conference*. New York, USA.

64. Nickelson, D. W. (1998). Telehealth and the Evolving Health Care System: Strategic Opportunities for Professional Psychology. *Professional Psychology: Research and Practice*, 29(6), 527-535.

65. Notice of Security Breach State Laws (2007). Retrieved May 30, 2008, from

http://www.consumersunion.org/campaigns/Breach_laws_May05.pdf

66. Noumeir, R., and Chafik, A. (2005). Access control and confidentiality in radiology. *Proceedings of the Medical Imaging 2005*. San Diego, CA, USA.

67. Ostrov, B. F. (2005). 140 Kaiser patients' private data put online. *Mercury News*. Retrieved May 29, 2008, from

    http://seclists.org/isn/2005/Mar/0070.html

68. Payne, C. (2002). On the Security of Open Source Software. *Information Systems Journal*, 12(1), 61-78.

69. PeBenito, C., Mayer, F., and MacMillan, K. (2006). Reference Policy for Security Enhanced Linux. *Proceeding of the 2006 Security Enhanced Linux Symposium*. Baltimore, Maryland, USA.

70. Popa, B. (2007). Firefox Extension Vulnerable to Attacks!. *SoftPedia*. Retrieved March 28, 2008, from

    http://news.softpedia.com/news/Firefox-Extension-Vulnerable-To-Attacks-46706.shtml

71. PRWeb (2007). *eHVRP Study Finds Healthcare Industry Must Do More to Protect Electronic Health Record Systems*. Retrieved May 16, 2008, from

    http://www.prwebdirect.com/releases/2007/9/prweb554028.php

72. Ray, P., Wimalasiri, J. (2006). The Need for Technical Solutions for Maintaining the Privacy of EHR. *Proceedings of the 28th IEEE EMBS Annual International Conference*. New York, USA.

73. Reid, J., Cheong, I., Henricksen, M., and Smith, J. (2003). A novel use of RBAC to protect privacy in distributed health care information systems. *Proceedings of the 8th Australasian Conference on Information Security and Privacy*. Wollongong, NSW, Australia.

74. Rojas-Burke, J. (2006, September). Providence settles data breach. *The Oregonian.*

75. Sahadi, J. (2005). 40M credit cards hacked. *CNN*. Retrieved May 18, 2008, from

    http://money.cnn.com/2005/06/17/news/master_card/index.htm

76. Saporito, B. (2005, June). The e-Health Revolution. *Time*. Retrieved April 14, 2008, from

    http://www.worldprivacyforum.org/pdf/pamdixonNCVHStestimonyfinal.pdf

77. Sayer, P. (2006, October). First security flaw signaled in IE7. *Computerworld.* Retrieved May 05, 2008, from

http://computerworld.com/action/article.do?command=viewArticleBasic&taxonomyName=viruses__worms_and_security_holes&articleId=9004259&taxonomyId=85

78. Schwartmann, D. (2004). An attributable Role-Based Access Control for Healthcare. *Proceedings of the 4th International Conference in Computer Science (ICCS 2004)*. Krakow, Poland.

79. Seaton, B. (2007, September). EHealth Vulnerability Reporting Program. *eHealthRisk*. Retrieved April 18, 2008, from

http://ehealthrisk.blogspot.com/2007/09/ehealth-vulnerability-reporting-program.html

80. SELinux Policy Server. (n.d.). *Tresys Technology*. Retrieved April 12, 2008, from

http://oss.tresys.com/projects/policy-server

81. SELinux Reference Policy. (n.d.). *Tresys Technology*. Retrieved May 30, 2008, from

http://oss.tresys.com/projects/refpolicy

82. *SELinux Symposium* (2008). Retrieved May 14, 2008, from

http://selinux-symposium.org/

83. Spencer, R., Smalley, S., Loscocco, P., Hibler, M., Andersen, D., and Lepreau, J. (1999). The flask security architecture: system support for diverse security policies. *Proceedings of the 8th Conference on USENIX Security Symposium*. Washington, D.C., USA.

84. Snyder, A. M., and Weaver, A. C. (2003). The e-logistic of securing distributed medical data. *Proceedings of the 1st IEEE International Conference on Industrial Informatics (INDIN'03)*. Banf, Alberta, Canada.

85. Summit Daily News (2005). Hackers breach CU computers. *The Associated Press*. Retrieved May 29, 2008, from

http://www.summitdaily.com/article/20050802/NEWS/50802002

86. Swedberg, C. (2007). Alzheimer's Care Center to Carry Out VeriChip Pilot. *RFID Journal*. Retrieved May 23, 2008, from

http://www.rfidjournal.com/article/articleprint/3340/-1/1/

87. Swedberg, C. (2008). VeriChip Markets Its Implantable RFID Tags and Services Direct to Consumers. *RFID Journal*. Retrieved May 23, 2008, from

    http://www.rfidjournal.com/article/articleprint/4055/-1/1/

88. Timbleby, H., Anderson, S., and Cairns, P. (1998). A Framework for Modelling Trojans and Computer Virus Infection. *The Computer Journal*, 41(7), 444-458.

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

90. *Unix Manual Page for profile* (n.d.). Retrieved May 20, 2008, from

    http://www.scit.wlv.ac.uk/cgi-bin/mansec?4+profile

91. Vamosi, R. (2007, September). Study finds electronic health records vulnerable. *CNET News*. Retrieved March 12, 2008, from

    http://www.news.com/8301-10784_3-9779986-7.html?tag=yt

92. Weaver, A. C., Dwyer, S. J., Snyder, A. M., Van Dyke, J., Hu, J., Chen, X., et al. (2003). Federated, secure trust networks for distributed healthcare IT services. *Proceedings of the First IEEE International Conference on Industrial Informatics*. Banf, Alberta, Canada.

93. Weiss, G. J. (2008). Open-Source Software in the Server OS Market, 2008: The State of Linux. *Gartner*.

94. Westhues, J. (2006, July). *Demo: Cloning a VeriChip*. Retrieved April 10, 2008, from

    http://cq.cx/verichip.pl

95. Williams, B. (2007). SLIDE: The SELinux Policy IDE. *Proceedings of the 2007 SELinux Symposium*. Baltimore, Maryland.

96. Win, K. T., Susilo, W., and Mu, Y. (2006). Personal Health Record System and Their Security Protection. *Journal of Medical Systems*, 30(4), 309-315.

97. Wright, C., Cowan, C., Morris, J., Smalley, S., and Kroah-Hartman, G. (2002). Linux Security Module Framework. *Proceeding of the Ottawa Linux Symposium*. Ottawa, Canada.

98. Zwillinger, M. J., and Sadker, J. (2005). Complying with breach notification laws. *Compliance and Governance Digest*. Retrieved May 28, 2008, from

http://searchfinancialsecurity.techtarget.com/tip/0,289483,sid185_gci1294335,00.html

Appendices

Appendix A

This is the list of acronyms used throughout this thesis.

Access Control List	ACL
Access Vector Cache	AVC
Access Vector Rule	AVR
Advance Encryption Standard	AES
American Hospital Association	AHA
Assembly Bill (AB)	AB
Australian Capital Territory	ACT
Automatic Internal Cardiac Defibrillator	AICD
Commercial IP Security Option	CIPSO
Common Criteria for Information Technology Security	CC
Common Desktop Environment	CDE
Confidentiality, Integrity and Availability Principles	CIA
Constrained Data Item	CDI
Context Based Access Control	CBAC
Control Access Protection Profile	CAPP
Data Encryption Standard	DES
Database	DB
Denial of Service	DoS
Department of Defence	DoD
Discretionary Access Control	DAC
Distributed Trusted Mach	DTMach
Distributed Trusted Operating System	DTOS
Domain and Type Enforcement	DTE
Domain Definition Table	DDT
Domain Interaction Table	DIT
Dynamic Policy Enforcement Agent	DEPA
eHealth Vulnerability Reporting Program	eHVRP
Electronic Health Record	EHR

Elliptic Curve Cryptography	ECC
Evaluation Assurance Level	EAL
Evaluation Assurance Level 4	EAL4
Extended attributes	xattrs
Flux Advanced Security Kernel	Flask
General Electric	GE
General Practitioner	GP
Global Positioning Systems	GPS
GNU’s Not Unix	GNU
Graphic User Interface	GUI
Group Identifier	GID
Health and Human Services	HHS
Health Information Systems	HIS
Health Insurance Portability and Accountability Act	HIPAA
Health Records and Information Privacy Act 2002	HRIP Act
Hyper Text Transfer Protocol over Secure Socket Layer	HTTPS
Implantable Cardioverter-Defibrillator	ICD
Individually Identifiable Health Information	IIHI
Information and Communication Technologies	ICT
Information Privacy Principles	IPP
Integrated Health Record and Information Systems	IHRIS
Integrity Verification Procedures	IVP
Internet Explorer	IE
Interprocess Communication	IPC
Intrusion Detection System	IDS
Java Desktop System	JDS
Labelled Security Protection Profile	LSPP
Linux Security Module	LSM
Mandatory Access Control	MAC
Massachusetts Institute of Technology	MIT

Microsoft	MS
Multi Level Category	MLC
Multi Level Security	MLS
National Health System	NHS
National Heart, Lung and Blood Institute	NHLBI
National Health and Medical Research Council	NHMRC
National Institute of Standard and Technology	NIST
National Privacy Principles	NPP
National Provider Identifier	NPI
National Security Agency	NSA
Nationwide Health Information Network	NHIN
Open-Source Software	OSS
Open Trusted Health Informatics Structure	OTHIS
Operating System	OS
Personal Identification Number	PIN
Personal Information Protection Act 1998	PPOP Act
Protected Health Information	PHI
Protection Profiles :	PP
Public Key Infrastructure	PKI
Queensland University of Technology	QUT
Radio-Frequency Identification	RFID
Red Hat Enterprise Linux	RHEL
Red Hat Enterprise Linux V5	RHEL5
Rivest-Shamir-Adleman	RSA
Role-Based Access Control	RBAC
Role-Based Access Protection Profile	RBACPP
Secure Computing Corporation	SCC
Secure Socket Layer	SSL
Security Enhanced Linux	SELinux
Security Identifier	SID

Security Target	ST
Separation of Duties	SoD
Social Security Number	SSN
Software Development Kit	SDK
Target of Evaluation	TOE
Total Cost of Ownership	TCO
Transformation Procedure	TP
Transport Layer Security	TLS
Triple-DES	3DES
Trusted Computer System Evaluation Criteria	TCSEC
Type Enforcement	TE
Unconstrained Data Item	UDI
University of California, Los Angles	UCLA
User Identifier	UID
Victorian Information Privacy Act 2000	VIP Act

Appendix B

Loadable Policy Module for the Diagnostic Application

The following Reference Policy module corresponds to the loadable policy module of the *Diagnostic Application*.

This is the content of the diag_sys.te file.

```
policy_module(diag_sys,1.4.3)


########################################
#
# Declarations
#

## <desc>
## <p>
## Allow users to access protected resources of the Diagnostic Application in case of
## emergency.
## </p>
## </desc>
gen_tunable(hc_diag_emerg,false)

# Type for the Diagnostic system executable file
type hc_diag_sys_exec_t;
files_type(hc_diag_sys_exec_t)

# Definition of the Diagnostic Application directory type
type hc_res_dbdir_t;
files_type(hc_res_dbdir_t)

# Definition of the Diagnostic Application files type for researchers
type hc_res_dbfile_t;
files_type(hc_res_dbfile_t)

# Definition of the Diagnostic Application diagnostic files type for researchers
type hc_res_dbfile_di_t;
files_type(hc_res_dbfile_di_t)

# Definition of the Diagnostic Application diagnostic files type for patients
type hc_pnt_dbfile_di_t;
files_type(hc_pnt_dbfile_di_t)

# Create the authorised roles and types for users and grant them access to their home
# directory.
healthcare_create_users(hc_doc)
healthcare_create_users(hc_pnt)
healthcare_create_users(hc_nur)
healthcare_create_users(hc_res)
healthcare_create_users(hc_locgp)

# Creates domains for the Diagnostic Application while executed by doctors and
# authorise doctors to access this domain.
```

```
diag_general_domain(hc_doc)

# Creates domains for the Diagnostic Application while executed by patients and
# authorise patients to access this domain.
diag_general_domain(hc_pnt)

# Creates domains for the Diagnostic Application while executed by nurses and
# authorise nurses to access this domain.
diag_general_domain(hc_nur)

# Creates domains for the Diagnostic Application while executed by researchers and
# authorise researchers to access this domain.
diag_general_domain(hc_res)

# Creates domains for the Diagnostic Application while executed by researchers and
# authorise researchers to access this domain.
diag_general_domain(hc_locgp)

# Authorise doctors to create diagnostic files through the use of the Diagnostic
# Application domain for doctors.
diag_creating_domain(hc_doc)

# Authorise doctors, nurses and patients to check diagnostic files through the use of
# their respective Diagnostic Application domains.
diag_check_domain(hc_doc)
diag_check_domain(hc_nur)
diag_check_domain(hc_pnt)

# Authorise researchers to check diagnostic files through the use of
# the Diagnostic Application domain for researchers.
diag_restcheck_domain(hc_res)

# Conditional policy which authorise the local GP user to check diagnostic files
# through the use of the Diagnostic Application domain for the local GPs.
tunable_policy(`hc_diag_emerg',`
	diag_check_domain(hc_locgp)
')

# Role declaration. Authorise the roles to access the domains of the Diagnostic
# Application.
role hc_doc_r types { hc_doc_diag_t };
role hc_pnt_r types { hc_pnt_diag_t };
role hc_nur_r types { hc_nur_diag_t };
role hc_res_r types { hc_res_diag_t };
role hc_locgp_r types { hc_locgp_diag_t };

allow system_r hc_doc_r;
allow system_r hc_pnt_r;
allow system_r hc_nur_r;
allow system_r hc_res_r;
allow system_r hc_locgp_r;
```

This is the content of the diag_sys.if file.

```
## <summary>
##        Diagnostic Application. The application is used to create and check
##        diagnostic reports of the patients.
## </summary>

########################################
## <summary>
##  This interface creates a domain that is used
##        by the diagnostic system.
## </summary>
## <param name="type">
##        <summary>
##        The prefix to be used by the domain.
##        </summary>
## </param>
#
interface(`diag_general_domain',`
        type $1_diag_t;
        domain_type($1_diag_t)

        domain_auto_trans($1_t, hc_diag_sys_exec_t, $1_diag_t)
        domain_entry_file($1_diag_t, hc_diag_sys_exec_t)

        allow $1_diag_t $1_t:process sigchld;
        allow $1_diag_t $1_tty_device_t:chr_file { rw_term_perms append };
        allow $1_diag_t $1_devpts_t:chr_file { rw_term_perms append };

# Access to shared libraries
        libs_use_ld_so($1_diag_t)
        libs_use_shared_libs($1_diag_t)

        miscfiles_read_localization($1_diag_t)

        init_read_utmp($1_diag_t)

        allow $1_diag_t hc_topdir_t:dir { search_dir_perms };
        allow $1_diag_t hc_topdir_db_t:dir { search_dir_perms };
')

########################################
## <summary>
##  This interface allow the creation of diagnostics for
##        the specified domain prefix.
## </summary>
## <param name="type">
##        <summary>
##        The prefix of the domain to gran creation access.
##        </summary>
## </param>
#
interface(`diag_creating_domain',`
allow $1_diag_t hc_pnt_dbdir_t:dir { add_entry_dir_perms create_dir_perms };
```

```
	allow $1_diag_t hc_pnt_dbdir_usr_t:dir { add_entry_dir_perms
create_dir_perms };

	type_transition $1_diag_t hc_pnt_dbdir_t:dir hc_pnt_dbdir_usr_t;

	allow $1_diag_t hc_pnt_dbfile_di_t:file { create_file_perms rw_file_perms };

	type_transition $1_diag_t hc_pnt_dbdir_usr_t:file hc_pnt_dbfile_di_t;

	allow $1_diag_t hc_res_dbdir_t:dir { create_dir_perms add_entry_dir_perms };
	allow $1_diag_t hc_res_dbfile_di_t:file { create_file_perms rw_file_perms };

	type_transition $1_diag_t hc_res_dbdir_t:file hc_res_dbfile_di_t;

	healthcare_append_audit_domain($1_diag_t)
')

########################################
## <summary>
##  This interface allow a specific domain to check
##	the diagnostics.
## </summary>
## <param name="type">
##	<summary>
##	The prefix of the domain to gran read access.
##	</summary>
## </param>
#
interface(`diag_check_domain',`
	allow $1_diag_t hc_pnt_dbdir_t:dir { search_dir_perms };
	allow $1_diag_t hc_pnt_dbdir_usr_t:dir { search_dir_perms };
	allow $1_diag_t hc_pnt_dbfile_di_t:file { read_file_perms };

	healthcare_append_audit_domain($1_diag_t)
')

########################################
## <summary>
##  This interface allow a specific domain to check
##	the diagnostics with restrictions.
## </summary>
## <param name="type">
##	<summary>
##	The prefix of the domain to gran restricted read access.
##	</summary>
## </param>
#
interface(`diag_restcheck_domain',`
	allow $1_diag_t hc_res_dbdir_t:dir { search_dir_perms };
	allow $1_diag_t hc_res_dbfile_di_t:file { read_file_perms };

	healthcare_append_audit_domain($1_diag_t)
')
```

This is the content of the diag_sys.fc file

```
#
# /healthcare/db
#
/healthcare/db/researchers/.*\.di       --
            gen_context(system_u:object_r:hc_res_dbfile_di_t,s0)
/healthcare/db/patients/.*/.*\.di              --
            gen_context(system_u:object_r:hc_pnt_dbfile_di_t,s0)

#
# /bin
#
/bin/diag_sys                                                  --
            gen_context(system_u:object_r:hc_diag_sys_exec_t,s0)
```

Appendix C

Loadable Policy Module for the Healthcare System Prototype

The following Reference Policy module corresponds to the loadable policy module of the complete prototype of this research. In this prototype simple applications were created and protected against malicious code and/or uses.

The content of the health_sys.te is as follows:

```
policy_module(health_sys,2.1.3)

########################################
#
# Declarations
#

## <desc>
## <p>
## Allow users to access the diagnosis system in case of emergency
## </p>
## </desc>
gen_tunable(hc_diag_emerg,false)

# Type for the appointment executable
type hc_appmt_sys_exec_t;
files_type(hc_appmt_sys_exec_t)

# Type for the diagnostic executable
type hc_diag_sys_exec_t;
files_type(hc_diag_sys_exec_t)

# Type for the Medication System executable
type hc_med_sys_exec_t;
files_type(hc_med_sys_exec_t)

# Type for the cash register system executable
type hc_cash_regi_exec_t;
files_type(hc_cash_regi_exec_t)

# Types for the application
type hc_topdir_t;
files_type(hc_topdir_t)

type hc_topdir_db_t;
files_type(hc_topdir_db_t)

# Typse for the audit part of the system
type hc_topdir_audit_t;
files_type(hc_topdir_audit_t)

type hc_audit_file_log_t;
files_type(hc_audit_file_log_t)
```

```
type hc_audit_dbdir_t;
files_type(hc_audit_dbdir_t)

type hc_audit_dbfile_t;
files_type(hc_audit_dbfile_t)

type hc_audit_dbfile_log_t;
files_type(hc_audit_dbfile_log_t)

type hc_audit_fdbdir_t;
files_type(hc_audit_fdbdir_t)

type hc_audit_fdbfile_log_t;
files_type(hc_audit_fdbfile_log_t)

# Types for the Appointment System
healthcare_files_dirs_template(hc_doc,app)
healthcare_files_dirs_template(hc_pnt,app)

type hc_doc_dbdir_tmp_t;
files_type(hc_doc_dbdir_tmp_t)

type hc_doc_dbfile_tmp_t;
files_type(hc_doc_dbfile_tmp_t)

# Types for the Diagnostic System
type hc_res_dbdir_t;
files_type(hc_res_dbdir_t)

type hc_res_dbfile_t;
files_type(hc_res_dbfile_t)

type hc_res_dbfile_di_t;
files_type(hc_res_dbfile_di_t)

type hc_pnt_dbfile_di_t;
files_type(hc_pnt_dbfile_di_t)

# Types for the Medication System
healthcare_files_dirs_template(hc_pha,mp)

type hc_res_dbfile_mp_t;
files_type(hc_res_dbfile_mp_t)

type hc_pnt_dbfile_mp_t;
files_type(hc_pnt_dbfile_mp_t)

# Types for the Cash Register System
type hc_topdir_findb_t;
files_type(hc_topdir_findb_t)

cash_reg_files_dirs_template(hc_fmgr,cr)
cash_reg_files_dirs_template(hc_adm,cr)
cash_reg_files_dirs_template(hc_final,cr)
```

```
# Create the healthcare users
healthcare_create_users(hc_doc)
healthcare_create_users(hc_pnt)
healthcare_create_users(hc_adm)
healthcare_create_users(hc_nur)
healthcare_create_users(hc_res)
healthcare_create_users(hc_pha)

healthcare_create_users(hc_locgp)

healthcare_create_users(hc_imgr)
healthcare_create_users(hc_audit)

healthcare_create_users(hc_fmgr)

# For the appointment
appmnt_doctor_domain(hc_doc)
appmnt_pnt_domain(hc_pnt)
appmnt_admin_domain(hc_adm)

# For the diagnostic
diag_general_domain(hc_doc)
diag_general_domain(hc_pnt)
diag_general_domain(hc_nur)
diag_general_domain(hc_res)

# Access to the diag_sys domains for the local GP user
diag_general_domain(hc_locgp)

diag_creating_domain(hc_doc)

diag_check_domain(hc_doc)
diag_check_domain(hc_nur)
diag_check_domain(hc_pnt)

tunable_policy(`hc_diag_emerg',`
	diag_check_domain(hc_locgp)
')

diag_restcheck_domain(hc_res)

# For the Medication System
med_general_domain(hc_doc)
med_general_domain(hc_pnt)
med_general_domain(hc_nur)
med_general_domain(hc_res)
med_general_domain(hc_pha)
med_general_domain(hc_locgp)

med_creating_domain(hc_doc)

med_check_domain(hc_doc,hc_pnt)
med_check_domain(hc_nur,hc_pnt)
med_check_domain(hc_pnt,hc_pnt)
med_check_domain(hc_pha,hc_pha)

med_restcheck_domain(hc_res)

med_delete_domain(hc_doc,hc_pnt)
med_delete_domain(hc_pha,hc_pha)

tunable_policy(`hc_diag_emerg',`
        med_creating_domain(hc_locgp)
        med_check_domain(hc_locgp,hc_pnt)
')

# For the Cash Register System
cash_regi_general_domain(hc_fmgr)
cash_regi_general_domain(hc_adm)

cash_regi_create_amount_domain(hc_fmgr)
cash_regi_create_amount_domain(hc_adm)

cash_regi_commit_domain(hc_fmgr)

# Permisison to allow access to the hc_audit_t
allow hc_audit_t hc_topdir_t:dir { list_dir_perms };
allow hc_audit_t hc_topdir_db_t:dir { list_dir_perms };
allow hc_audit_t hc_topdir_findb_t:dir { list_dir_perms };

healthcare_read_audit_domain(hc_audit_t)

# Permisisons to allow access to the hc_imgr_t
allow hc_imgr_t hc_topdir_t:dir { list_dir_perms };
allow hc_imgr_t hc_topdir_db_t:dir { list_dir_perms };

healthcare_read_pntinfo_domain(hc_imgr_t)


# Roles declaration
role hc_doc_r types { hc_doc_app_t hc_doc_diag_t hc_doc_med_t };
role hc_adm_r types { hc_adm_app_t hc_adm_cr_t };
role hc_pnt_r types { hc_pnt_app_t hc_pnt_diag_t hc_pnt_med_t };
role hc_nur_r types { hc_nur_diag_t hc_nur_med_t };
role hc_res_r types { hc_res_diag_t hc_res_med_t };
role hc_pha_r types { hc_pha_med_t };
role hc_fmgr_r types { hc_fmgr_cr_t };

role hc_locgp_r types { hc_locgp_diag_t hc_locgp_med_t };

allow system_r hc_doc_r;
allow system_r hc_adm_r;
allow system_r hc_pnt_r;
allow system_r hc_nur_r;
allow system_r hc_res_r;
allow system_r hc_pha_r;
allow system_r hc_locgp_r;
allow system_r hc_fmgr_r;
```

The content of the health_sys.if is as follows:

```
## <summary>
##	Healchtcare System. This system is responsible for the
##	creation of appointments, diagnostic reports, audit reports,
##	medication requests, and cash checks.
## </summary>

########################################
## <summary>
##  Creates the types for files and directories to be used
##	in the appointment system.
## </summary>
## <param name="type">
##	<summary>
##	The type prefix to create.
##	</summary>
## </param>
## <param name="type">
##	<summary>
##	The name of the application file.
##	</summary>
## </param>
#
template(`healthcare_files_dirs_template',`
	type $1_dbdir_t;
	files_type($1_dbdir_t)

	type $1_dbfile_t;
	files_type($1_dbfile_t)

	type $1_dbdir_usr_t;
	files_type($1_dbdir_usr_t)

	type $1_dbfile_$2_t;
	files_type($1_dbfile_$2_t)
')

########################################
## <summary>
##  This interface allows to create a type
##	with general caracteristics for the
##	use of the appointment system.
## </summary>
## <param name="type">
##	<summary>
##	The type prefix to create.
##	</summary>
## </param>
#
interface(`appmnt_general_domain',`
	type $1_app_t;
	domain_type($1_app_t)

	domain_auto_trans($1_t, hc_appmt_sys_exec_t, $1_app_t)
	domain_entry_file($1_app_t, hc_appmt_sys_exec_t)

	allow $1_app_t $1_t:process sigchld;
	allow $1_app_t $1_tty_device_t:chr_file { rw_term_perms append };
	allow $1_app_t $1_devpts_t:chr_file { rw_term_perms append };

# Access to shared libraries
	libs_use_ld_so($1_app_t)
	libs_use_shared_libs($1_app_t)

	miscfiles_read_localization($1_app_t)
	kernel_read_system_state($1_app_t)

	init_read_utmp($1_app_t)

	allow $1_app_t hc_topdir_t:dir { search_dir_perms };
	allow $1_app_t hc_topdir_db_t:dir { search_dir_perms };
')

########################################
## <summary>
##	Allows the specified type to create appointments
##	in the system.
## </summary>
## <param name="type">
##	<summary>
##	Type that is allowed to create.
##	</summary>
## </param>
#
interface(`appmnt_creating_domain',`
	allow $1_app_t hc_doc_dbdir_t:dir { create_dir_perms add_entry_dir_perms };
	allow $1_app_t hc_doc_dbdir_usr_t:dir { create_file_perms
add_entry_dir_perms };
	allow $1_app_t hc_doc_dbfile_app_t:file { append_file_perms
create_file_perms rw_file_perms };

	type_transition $1_app_t hc_doc_dbdir_t:dir hc_doc_dbdir_usr_t;
	type_transition $1_app_t hc_doc_dbdir_usr_t:file hc_doc_dbfile_app_t;


	allow $1_app_t hc_pnt_dbdir_t:dir { create_dir_perms add_entry_dir_perms };
	allow $1_app_t hc_pnt_dbdir_usr_t:dir { create_dir_perms
add_entry_dir_perms };
	allow $1_app_t hc_pnt_dbfile_app_t:file { append_file_perms
create_file_perms rw_file_perms };

	type_transition $1_app_t hc_pnt_dbdir_t:dir hc_pnt_dbdir_usr_t;
	type_transition $1_app_t hc_pnt_dbdir_usr_t:file hc_pnt_dbfile_app_t;

	healthcare_append_audit_domain($1_app_t)
')
```

```
########################################
## <summary>
##      Allows the specified type to authorize appointments
##      in the system.
## </summary>
## <param name="type">
##      <summary>
##      Type to be used to authorize.
##      </summary>
## </param>
#
interface(`appmnt_authorize_domain',`
        allow $1_app_t hc_doc_dbdir_t:dir { search_dir_perms };
        allow $1_app_t hc_doc_dbdir_usr_t:dir { search_dir_perms };

        allow $1_app_t hc_doc_dbdir_tmp_t:dir { search_dir_perms
del_entry_dir_perms };
        allow $1_app_t hc_doc_dbfile_tmp_t:file { read_file_perms delete_file_perms
};

        healthcare_append_audit_domain($1_app_t)
')

########################################
## <summary>
##      Allows the specified type to check appointments
##      in the system.
## </summary>
## <param name="type">
##      <summary>
##      Type that is allowed to access.
##      </summary>
## </param>
## <param name="type">
##      <summary>
##      Type to be accessed.
##      </summary>
## </param>
#
interface(`appmnt_check_domain',`
        allow $1_app_t $2_dbdir_t:dir { search_dir_perms };
        allow $1_app_t $2_dbdir_usr_t:dir { search_dir_perms };
        allow $1_app_t $2_dbfile_app_t:file { read_file_perms };

        healthcare_append_audit_domain($1_app_t)
')

########################################
## <summary>
##      Allows the specified type to delete appointments
##      in the system.
## </summary>
## <param name="type">
##      <summary>
##        Type that is allowed to delete.
##        </summary>
## </param>
## <param name="type">
##        <summary>
##        Type to be deleted.
##        </summary>
## </param>
#
interface(`appmnt_delete_domain',`
          allow $1_app_t $2_dbdir_t:dir { search_dir_perms };
          allow $1_app_t $2_dbdir_usr_t:dir { search_dir_perms del_entry_dir_perms };
          allow $1_app_t $2_dbfile_app_t:file { delete_file_perms };

          healthcare_append_audit_domain($1_app_t)
')

########################################
## <summary>
##        Allows the specified type enter the doctor domain for the
##  creation of appointments.
## </summary>
## <param name="type">
##        <summary>
##        Type to be used as the doctor type.
##        </summary>
## </param>
#
interface(`appmnt_doctor_domain',`
          appmnt_general_domain($1)

          appmnt_creating_domain($1)

          appmnt_authorize_domain($1)

          appmnt_check_domain($1,$1)

          appmnt_delete_domain($1,$1)
')

########################################
## <summary>
##        Allow the specified type to enter the admin staff domain
##        for the creation of appointments.
## </summary>
## <param name="type">
##        <summary>
##        Type to be used as the admin staff type.
##        </summary>
## </param>
#
interface(`appmnt_admin_domain',`
          appmnt_general_domain($1)

          allow $1_app_t hc_doc_dbdir_t:dir { add_entry_dir_perms create_dir_perms };
```

```
	type_transition $1_app_t hc_doc_dbdir_t:dir hc_doc_dbdir_usr_t;

	allow $1_app_t hc_doc_dbdir_usr_t:dir { add_entry_dir_perms
create_dir_perms };

	type_transition $1_app_t hc_doc_dbdir_usr_t:dir hc_doc_dbdir_tmp_t;

	allow $1_app_t hc_doc_dbdir_tmp_t:dir { add_entry_dir_perms
create_dir_perms };
	allow $1_app_t hc_doc_dbfile_tmp_t:file { append_file_perms
create_file_perms rw_file_perms };

	type_transition $1_app_t hc_doc_dbdir_tmp_t:file hc_doc_dbfile_tmp_t;

	appmnt_check_domain($1,hc_pnt)

	appmnt_delete_domain($1,hc_pnt)
')

########################################
## <summary>
##	Allow the specified type to enter the patient domain
##	for the use of the appointments system.
## </summary>
## <param name="type">
##	<summary>
##	Type to be used as prefix for the domain.
##	</summary>
## </param>
#
interface(`appmnt_pnt_domain',`
	appmnt_general_domain($1)

	appmnt_check_domain($1,$1)
')

########################################
## <summary>
##	Create users that are required in the healthcare system.
## </summary>
## <param name="type">
##	<summary>
##	Type to be used as prefix for the users and roles.
##	</summary>
## </param>
#
interface(`healthcare_create_users',`
	userdom_unpriv_user_template($1)

	corecmd_shell_entry_type($1_t)
	corecmd_exec_shell($1_t)

	userdom_read_generic_user_home_content_files($1_t)
	userdom_manage_generic_user_home_content_dirs($1_t)

	allow $1_t user_home_dir_t:file { manage_file_perms };
')

########################################
## <summary>
##  This interface creates a domain that is used
##	by the diagnostic system.
## </summary>
## <param name="type">
##	<summary>
##	The prefix to be used by the domain.
##	</summary>
## </param>
#
interface(`diag_general_domain',`
	type $1_diag_t;
	domain_type($1_diag_t)

	domain_auto_trans($1_t, hc_diag_sys_exec_t, $1_diag_t)
	domain_entry_file($1_diag_t, hc_diag_sys_exec_t)

	allow $1_diag_t $1_t:process sigchld;
	allow $1_diag_t $1_tty_device_t:chr_file { rw_term_perms append };
	allow $1_diag_t $1_devpts_t:chr_file { rw_term_perms append };

# Access to shared libraries
	libs_use_ld_so($1_diag_t)
	libs_use_shared_libs($1_diag_t)

	miscfiles_read_localization($1_diag_t)

	init_read_utmp($1_diag_t)

	allow $1_diag_t hc_topdir_t:dir { search_dir_perms };
	allow $1_diag_t hc_topdir_db_t:dir { search_dir_perms };
')

########################################
## <summary>
##  This interface allow the creation of diagnostics for
##	the specified domain prefix.
## </summary>
## <param name="type">
##	<summary>
##	The prefix of the domain to gran creation access.
##	</summary>
## </param>
#
interface(`diag_creating_domain',`
	allow $1_diag_t hc_pnt_dbdir_t:dir { add_entry_dir_perms create_dir_perms };
	allow $1_diag_t hc_pnt_dbdir_usr_t:dir { add_entry_dir_perms
create_dir_perms };
```

```
        type_transition $1_diag_t hc_pnt_dbdir_t:dir hc_pnt_dbdir_usr_t;

        allow $1_diag_t hc_pnt_dbfile_di_t:file { create_file_perms rw_file_perms };

        type_transition $1_diag_t hc_pnt_dbdir_usr_t:file hc_pnt_dbfile_di_t;

        allow $1_diag_t hc_res_dbdir_t:dir { create_dir_perms add_entry_dir_perms };
        allow $1_diag_t hc_res_dbfile_di_t:file { create_file_perms rw_file_perms };

        type_transition $1_diag_t hc_res_dbdir_t:file hc_res_dbfile_di_t;

        healthcare_append_audit_domain($1_diag_t)
')

########################################
## <summary>
##  This interface allow a specific domain to check
##      the diagnostics.
## </summary>
## <param name="type">
##      <summary>
##      The prefix of the domain to gran read access.
##      </summary>
## </param>
#
interface(`diag_check_domain',`
        allow $1_diag_t hc_pnt_dbdir_t:dir { search_dir_perms };
        allow $1_diag_t hc_pnt_dbdir_usr_t:dir { search_dir_perms };
        allow $1_diag_t hc_pnt_dbfile_di_t:file { read_file_perms };

        healthcare_append_audit_domain($1_diag_t)
')

########################################
## <summary>
##  This interface allow a specific domain to check
##      the diagnostics with restrictions.
## </summary>
## <param name="type">
##      <summary>
##      The prefix of the domain to gran restricted read access.
##      </summary>
## </param>
#
interface(`diag_restcheck_domain',`
        allow $1_diag_t hc_res_dbdir_t:dir { search_dir_perms };
        allow $1_diag_t hc_res_dbfile_di_t:file { read_file_perms };

        healthcare_append_audit_domain($1_diag_t)
')

########################################
## <summary>
##  This interface creates a domain that is used
##      by the medication system.
## </summary>
## <param name="type">
##        <summary>
##        The prefix to be used by the domain.
##        </summary>
## </param>
#
interface(`med_general_domain',`
        type $1_med_t;
        domain_type($1_med_t)

        domain_auto_trans($1_t, hc_med_sys_exec_t, $1_med_t)
        domain_entry_file($1_med_t, hc_med_sys_exec_t)

        allow $1_med_t $1_t:process sigchld;
        allow $1_med_t $1_tty_device_t:chr_file { rw_term_perms append };
        allow $1_med_t $1_devpts_t:chr_file { rw_term_perms append };

# Access to shared libraries
        libs_use_ld_so($1_med_t)
        libs_use_shared_libs($1_med_t)

        miscfiles_read_localization($1_med_t)

        init_read_utmp($1_med_t)

        allow $1_med_t hc_topdir_t:dir { search_dir_perms };
        allow $1_med_t hc_topdir_db_t:dir { search_dir_perms };
')

########################################
## <summary>
##  This interface allow the creation of medical precriptions
##  within the medical system domain.
## </summary>
## <param name="type">
##        <summary>
##        The prefix of the domain to grant creation access.
##        </summary>
## </param>
#
interface(`med_creating_domain',`

        allow $1_med_t hc_pnt_dbdir_t:dir { add_entry_dir_perms };
        allow $1_med_t hc_pnt_dbdir_usr_t:dir { add_entry_dir_perms
create_dir_perms };

        type_transition $1_med_t hc_pnt_dbdir_t:dir hc_pnt_dbdir_usr_t;

        allow $1_med_t hc_pnt_dbfile_mp_t:file { create_file_perms rw_file_perms };

        type_transition $1_med_t hc_pnt_dbdir_usr_t:file hc_pnt_dbfile_mp_t;

        allow $1_med_t hc_pha_dbdir_t:dir { add_entry_dir_perms };
```

```
	allow $1_med_t hc_pha_dbdir_usr_t:dir { add_entry_dir_perms
create_dir_perms };

	type_transition $1_med_t hc_pha_dbdir_t:dir hc_pha_dbdir_usr_t;

	allow $1_med_t hc_pha_dbfile_mp_t:file { create_file_perms rw_file_perms };

	type_transition $1_med_t hc_pha_dbdir_usr_t:file hc_pha_dbfile_mp_t;

	allow $1_med_t hc_res_dbdir_t:dir { add_entry_dir_perms };
	allow $1_med_t hc_res_dbfile_mp_t:file { create_file_perms rw_file_perms };

	type_transition $1_med_t hc_res_dbdir_t:file hc_res_dbfile_mp_t;

	healthcare_append_audit_domain($1_med_t)
')

########################################
## <summary>
##  This interface allow a specific domain to check
##	the medical precriptions of a specific type.
## </summary>
## <param name="type">
##	<summary>
##	The prefix of the domain to gran read access.
##	</summary>
## </param>
## <param name="type">
##	<summary>
##	The prefix of the types to access.
##	</summary>
## </param>
#
interface(`med_check_domain',`
	allow $1_med_t $2_dbdir_t:dir { search_dir_perms };
	allow $1_med_t $2_dbdir_usr_t:dir { search_dir_perms };
	allow $1_med_t $2_dbfile_mp_t:file { read_file_perms };

	healthcare_append_audit_domain($1_med_t)
')

########################################
## <summary>
##  This interface allow a specific domain to check
##	the medical prescriptions with restrictions.
## </summary>
## <param name="type">
##	<summary>
##	The prefix of the domain to gran restricted read access.
##	</summary>
## </param>
#
interface(`med_restcheck_domain',`
	allow $1_med_t hc_res_dbdir_t:dir { search_dir_perms };
	allow $1_med_t hc_res_dbfile_mp_t:file { read_file_perms };
```

```
        healthcare_append_audit_domain($1_med_t)
')

########################################
## <summary>
##      Allows the specified type to delete medical
##      prescriptions in the system.
## </summary>
## <param name="type">
##      <summary>
##      Type that is allowed to delete.
##      </summary>
## </param>
## <param name="type">
##      <summary>
##      Type to be deleted.
##      </summary>
## </param>
#
interface(`med_delete_domain',`
        allow $1_med_t $2_dbdir_t:dir { search_dir_perms };
        allow $1_med_t $2_dbdir_usr_t:dir { search_dir_perms del_entry_dir_perms };
        allow $1_med_t $2_dbfile_mp_t:file { delete_file_perms };

        healthcare_append_audit_domain($1_med_t)
')

########################################
## <summary>
##  This interface allows access to audit files and
##      directories.
## </summary>
## <param name="type">
##      <summary>
##      The type that is going to be allowed to access.
##      </summary>
## </param>
#
interface(`healthcare_read_audit_domain',`
        allow $1 hc_topdir_audit_t:dir { list_dir_perms };
        allow $1 hc_audit_file_log_t:file { read_file_perms };
        allow $1 hc_audit_dbdir_t:dir { list_dir_perms };
        allow $1 hc_audit_dbfile_log_t:file { read_file_perms };
        allow $1 hc_audit_fdbdir_t:dir { list_dir_perms };
        allow $1 hc_audit_fdbfile_log_t:file { read_file_perms };
')


########################################
## <summary>
##  This interface allows to append to audit files and
##      search access to audit directories.
## </summary>
## <param name="type">
```

```
##	<summary>
##	The type that is going to be allowed to access.
##	</summary>
## </param>
#
interface(`healthcare_append_audit_domain',`
	allow $1 hc_topdir_audit_t:dir { list_dir_perms };
	allow $1 hc_audit_file_log_t:file { append_file_perms };
	allow $1 hc_audit_dbdir_t:dir { list_dir_perms };
	allow $1 hc_audit_dbfile_log_t:file { append_file_perms };
')

########################################
## <summary>
##  This interface allows to append to audit files and
##	search access to audit directories.
## </summary>
## <param name="type">
##	<summary>
##	The type that is going to be allowed to access.
##	</summary>
## </param>
#
interface(`healthcare_append_fdbaudit_domain',`
	allow $1 hc_topdir_audit_t:dir { list_dir_perms };
	allow $1 hc_audit_file_log_t:file { append_file_perms };
	allow $1 hc_audit_fdbdir_t:dir { list_dir_perms };
	allow $1 hc_audit_fdbfile_log_t:file { append_file_perms };
')

########################################
## <summary>
##  Creates the types for files and directories to be used
##	in the cash register.
## </summary>
## <param name="type">
##	<summary>
##	The type prefix to create.
##	</summary>
## </param>
## <param name="type">
##	<summary>
##	The name of the application file.
##	</summary>
## </param>
#
template(`cash_reg_files_dirs_template',`
	type $1_fdbdir_t;
	files_type($1_fdbdir_t)

	type $1_fdbfile_t;
	files_type($1_fdbfile_t)

	type $1_fdbdir_usr_t;
	files_type($1_fdbdir_usr_t)
```

```
        type $1_fdbfile_$2_t;
        files_type($1_fdbfile_$2_t)
')

########################################
## <summary>
##  This interface creates a domain that is used
##      by the cash register system.
## </summary>
## <param name="type">
##      <summary>
##      The prefix to be used by the domain.
##      </summary>
## </param>
#
interface(`cash_regi_general_domain',`
        type $1_cr_t;
        domain_type($1_cr_t)

        domain_auto_trans($1_t, hc_cash_regi_exec_t, $1_cr_t)
        domain_entry_file($1_cr_t, hc_cash_regi_exec_t)

        allow $1_cr_t $1_t:process sigchld;
        allow $1_cr_t $1_tty_device_t:chr_file { rw_term_perms append };
        allow $1_cr_t $1_devpts_t:chr_file { rw_term_perms append };

# Access to shared libraries
        libs_use_ld_so($1_cr_t)
        libs_use_shared_libs($1_cr_t)

        miscfiles_read_localization($1_cr_t)

        init_read_utmp($1_cr_t)

        allow $1_cr_t hc_topdir_t:dir { search_dir_perms };
        allow $1_cr_t hc_topdir_findb_t:dir { search_dir_perms };
')

########################################
## <summary>
##  This interface allow the creation of cash registers.
## </summary>
## <param name="type">
##      <summary>
##      The prefix of the domain to grant creation access.
##      </summary>
## </param>
#
interface(`cash_regi_create_amount_domain',`
        allow $1_cr_t $1_fdbdir_t:dir { create_dir_perms add_entry_dir_perms };
        allow $1_cr_t $1_fdbdir_usr_t:dir { create_dir_perms add_entry_dir_perms };

        type_transition $1_cr_t $1_fdbdir_t:dir $1_fdbdir_usr_t;
```

```
	allow $1_cr_t $1_fdbfile_cr_t:file { create_file_perms rw_file_perms };

	type_transition $1_cr_t $1_fdbdir_usr_t:file $1_fdbfile_cr_t;

	healthcare_append_fdbaudit_domain($1_cr_t)
')

########################################
## <summary>
##  This interface allow to commit cash registers values.
## </summary>
## <param name="type">
##	<summary>
##	The prefix of the domain to grant creation access.
##	</summary>
## </param>
#
interface(`cash_regi_commit_domain',`
	allow $1_cr_t hc_fmgr_fdbdir_t:dir { list_dir_perms };
	allow $1_cr_t hc_fmgr_fdbdir_usr_t:dir { list_dir_perms };
	allow $1_cr_t hc_fmgr_fdbfile_cr_t:file { read_file_perms };

	allow $1_cr_t hc_adm_fdbdir_t:dir { list_dir_perms };
	allow $1_cr_t hc_adm_fdbdir_usr_t:dir { list_dir_perms };
	allow $1_cr_t hc_adm_fdbfile_cr_t:file { read_file_perms };

	allow $1_cr_t hc_final_fdbdir_t:dir { create_dir_perms add_entry_dir_perms };
	allow $1_cr_t hc_final_fdbdir_usr_t:dir { create_dir_perms
add_entry_dir_perms };

	type_transition $1_cr_t hc_final_fdbdir_t:dir hc_final_fdbdir_usr_t;

	allow $1_cr_t hc_final_fdbfile_cr_t:file { create_file_perms rw_file_perms };

	type_transition $1_cr_t hc_final_fdbdir_usr_t:file hc_final_fdbfile_cr_t;

	healthcare_append_fdbaudit_domain($1_cr_t)
')

########################################
## <summary>
##  This interface allows access to read patient
##	files.
## </summary>
## <param name="type">
##	<summary>
##	The type that is going to be allowed to access.
##	</summary>
## </param>
#
interface(`healthcare_read_pntinfo_domain',`
	allow $1 hc_pnt_dbdir_t:dir { list_dir_perms };
	allow $1 hc_pnt_dbdir_usr_t:dir { list_dir_perms };
	allow $1 hc_pnt_dbfile_app_t:file { read_file_perms };
	allow $1 hc_pnt_dbfile_di_t:file { read_file_perms };
```

```
        allow $1 hc_pnt_dbfile_mp_t:file { read_file_perms };
')
```

The content of the health_sys.fc is as follows:

```
#
# /healthcare
#

/healthcare                                             -d
        gen_context(system_u:object_r:hc_topdir_t,s0)
/healthcare/db                                          -d
        gen_context(system_u:object_r:hc_topdir_db_t,s0)
/healthcare/audit                                       -d
        gen_context(system_u:object_r:hc_topdir_audit_t,s0)
/healthcare/findb                                       -d
        gen_context(system_u:object_r:hc_topdir_findb_t,s0)

#
# /healthcare/audit
#

/healthcare/audit/.*\.log                   --
        gen_context(system_u:object_r:hc_audit_file_log_t,s0)

#
# /healthcare/db
#

/healthcare/db/doctors                              -d
        gen_context(system_u:object_r:hc_doc_dbdir_t,s0)
/healthcare/db/patients                             -d
        gen_context(system_u:object_r:hc_pnt_dbdir_t,s0)
/healthcare/db/researchers                          -d
        gen_context(system_u:object_r:hc_res_dbdir_t,s0)
/healthcare/db/pharmacists                          -d
        gen_context(system_u:object_r:hc_pha_dbdir_t,s0)
/healthcare/db/audit                                -d
        gen_context(system_u:object_r:hc_audit_dbdir_t,s0)

/healthcare/db/doctors(/.*)?                --
        gen_context(system_u:object_r:hc_doc_dbfile_t,s0)
/healthcare/db/patients(/.*)?               --
        gen_context(system_u:object_r:hc_pnt_dbfile_t,s0)
/healthcare/db/researchers(/.*)?  --
        gen_context(system_u:object_r:hc_res_dbfile_t,s0)
/healthcare/db/pharmacists(/.*)? --
        gen_context(system_u:object_r:hc_pha_dbfile_t,s0)
/healthcare/db/audit(/.*)?                          --
        gen_context(system_u:object_r:hc_audit_dbfile_t,s0)
```

```
/healthcare/db/doctors/.*                       -d
        gen_context(system_u:object_r:hc_doc_dbdir_usr_t,s0)
/healthcare/db/patients/.*                      -d
        gen_context(system_u:object_r:hc_pnt_dbdir_usr_t,s0)
/healthcare/db/pharmacists/.*           -d
        gen_context(system_u:object_r:hc_pha_dbdir_usr_t,s0)

/healthcare/db/audit/.*\.log            --
        gen_context(system_u:object_r:hc_audit_dbfile_log_t,s0)

/healthcare/db/doctors/.*/.*\.app       --
        gen_context(system_u:object_r:hc_doc_dbfile_app_t,s0)
/healthcare/db/patients/.*/.*\.app      --
        gen_context(system_u:object_r:hc_pnt_dbfile_app_t,s0)

/healthcare/db/researchers/.*\.di     --
        gen_context(system_u:object_r:hc_res_dbfile_di_t,s0)
/healthcare/db/patients/.*/.*\.di       --
        gen_context(system_u:object_r:hc_pnt_dbfile_di_t,s0)

/healthcare/db/researchers/.*\.mp     --
        gen_context(system_u:object_r:hc_res_dbfile_mp_t,s0)
/healthcare/db/patients/.*/.*\.mp       --
        gen_context(system_u:object_r:hc_pnt_dbfile_mp_t,s0)
/healthcare/db/pharmacists/.*/.*\.mp    --
        gen_context(system_u:object_r:hc_pha_dbfile_mp_t,s0)

/healthcare/db/doctors/.*/tmp                   -d
        gen_context(system_u:object_r:hc_doc_dbdir_tmp_t,s0)
/healthcare/db/doctors/.*/tmp/.*\.tmp   --
        gen_context(system_u:object_r:hc_doc_dbfile_tmp_t,s0)

#
# /healthcare/findb
#

/healthcare/findb/managers                              -d
        gen_context(system_u:object_r:hc_fmgr_fdbdir_t,s0)
/healthcare/findb/admins                                -d
        gen_context(system_u:object_r:hc_adm_fdbdir_t,s0)
/healthcare/findb/final                                 -d
        gen_context(system_u:object_r:hc_final_fdbdir_t,s0)
/healthcare/findb/audit                                 -d
        gen_context(system_u:object_r:hc_audit_fdbdir_t,s0)

/healthcare/findb/audit/.*\.log                 --
        gen_context(system_u:object_r:hc_audit_fdbfile_log_t,s0)

/healthcare/findb/managers(/.*)?        -d
        gen_context(system_u:object_r:hc_fmgr_fdbfile_t,s0)
/healthcare/findb/admins(/.*)?                  -d
        gen_context(system_u:object_r:hc_adm_fdbfile_t,s0)
/healthcare/findb/final(/.*)?                   -d
        gen_context(system_u:object_r:hc_final_fdbfile_t,s0)
```

```
/healthcare/findb/managers/.*                    -d
        gen_context(system_u:object_r:hc_fmgr_fdbdir_usr_t,s0)
/healthcare/findb/admins/.*                              -d
        gen_context(system_u:object_r:hc_adm_fdbdir_usr_t,s0)
/healthcare/findb/final/.*                               -d
        gen_context(system_u:object_r:hc_final_fdbdir_usr_t,s0)

/healthcare/findb/managers/.*/.*\.cr        -d
        gen_context(system_u:object_r:hc_fmgr_fdbfile_cr_t,s0)
/healthcare/findb/admins/.*/.*\.cr               -d
        gen_context(system_u:object_r:hc_adm_fdbfile_cr_t,s0)
/healthcare/findb/final/.*/.*\.cr           -d
        gen_context(system_u:object_r:hc_final_fdbfile_cr_t,s0)

#
# /bin
#

/bin/appmt_sys                                           --
        gen_context(system_u:object_r:hc_appmt_sys_exec_t,s0)
/bin/diag_sys                                            --
        gen_context(system_u:object_r:hc_diag_sys_exec_t,s0)
/bin/med_sys                                             --
        gen_context(system_u:object_r:hc_med_sys_exec_t,s0)
/bin/cash_regi                                           --
        gen_context(system_u:object_r:hc_cash_regi_exec_t,s0)
```

Appendix D

The following list briefly describes access permissions in SELinux as defined by Tresys Technologies.

- read. Read an object.
- write. Write an object.
- create. Open and create an object.
- getattr. Get the attributes of an object, such as stat().
- setattr. Modify the attributes of an object, such as chmod().
- append. Append only to an object.
- delete. Delete and object.
- manage. Create, read, write, and delete and object.
- relabelfrom. Relabel form the object type.
- relabelto. Relabel to the object type.
- relabel. Relabel to and from the object type
- exec. Execute a file in the callers domain (no domain transition; file only)
- search. Search a directory, but not get a list of directory entries.
- list. Read the list of directory entries.
- mounton. Filesystems can be mounted on this directory.
- sigchld. Send a SIGCHLD signal between processes.
- sigstop. Send a SIGSTOP signal between processes.
- signull. Send a null signal between processes.
- kill. Send a kill signal between processes.
- domtrans. Execute a program and perform a domain transition.