Security-Enhanced Linux in a Health Information System

Louiza Papachristodoulou, **Spyridon Antakis**

Eindhoven University of Technology
Department of Mathematic & Computer Science
Den Dolech 2, P.O Box 513, 5600 MB Eindhoven
Email: {l.p.papachristodoulou,s.antakis}@student.tue.nl

December 2, 2008

Abstract

This paper presents the security subsystem SELinux, which implements a flexible mandatory access control mechanism called Type Enforcement (TE), based on a security architecture called Flask. Furthermore, it analyzes basic features of the SELinux Policy Language and proposes its application in a Health Information System (HIS), in order to achieve the desired security level that is required to protect so sensitive databases. Finally, the advantages and disadvantages of the usage of SELinux in a HIS Infrastructure will be discussed and a conclusion on the efficiency of this subsystem will be made, concerning mostly security and privacy issues.

1 Introduction

The widespread adoption of Information Technology (IT) in healthcare systems all over the world [4], brings into the surface a lot of privacy related issues. This movement from the *traditional healthcare system* to a *new electronic healthcare system* depicts the urgent need of effective design and management of patient health information. *Electronic healthcare systems* aim to improve the quality of our life by providing to the medical staff the ability of remotely accessing medical records , usually through a public network, such as the Internet. Recording information electronically brings additional functionalities, such as the ability to deliver health information in real time to the point of care, when it is required for the purpose of assisting in clinical decision making and the reduction of medical errors.
Nevertheless, information stored within an electronic *Health Information System (HIS)* is highly sensitive by its nature. So, despite of all the benefits, electronic medical records create new security concerns. In this sense, *security*, meaning the protection of data *integrity, availability, authentication, confidentiality* and *privacy*, are critical factors towards achieving citizens trust and acceptance of health information systems. A security violation in a HIS requires an Operating System (OS), which can enforce Mandatory Access Control (MAC) rules, so that access to the resources does not rely on the discretion of the users. In this way, potential damages that arise from applications' compromisation, such as unauthorised disclosure or unauthorised alteration of individual health information could be minimized and thus any possible disaster among healthcare providers and consumers will be avoided. Several countries that have already developed electronic health services, such as Australia, the USA and the UK, had to overcome information privacy violations or weaknesses, which were found in their systems in the past.
The multi-user and resource-sharing environment, in which HIS services are provided, imposes the need to use the access control security mechanism, in order to protect the data resources. A modern HIS would normally consist of *health application services, middleware, Database Management System (DBMS), data network control system, operating system* and *hardware*. The main interest is to protect the patient's personal data, but the security in the application space cannot exist without particular security prerequisites at levels of *hardware, operating systems*

and any *middleware* sub-systems.
This paper focuses on the access control security mechanism (operating system level), and more precisely suggests using the *Security Enhanced Linux (SELinux)* policy subsystem to a HIS. In section 2, the *Flask Architecture* that SELinux uses, is briefly presented and described. Moreover, an introduction to *Linux Security Module (LSM)* is given, in order to make it easier for the reader to understand how SELinux enhances security in a Linux operating system. In section 3, some basic features of *SELinux Policy Languge* are presented and a short analysis of each feature is made. In section 4, mechanisms that SELinux could apply to a HIS are proposed, and the advantages and disadvantages of usage of SELinux security subsystem in a HIS are presented.

2 SELinux Mechanism

SELinux has its origins in an operating system security and microkernel research performed by the U.S. National Security Agency's research organization. All those efforts of research led to the creation of a new security architecture, called *Flask* . That architecture supported a more *flexible* and *dynamic type of enforcement* mechanism.[2] In the late 2000, the NSA released a *Flask* based security system in the Linux kernel with the name *Security Enhanced Linux*. At the same period, a *Linux Security Module (LSM)* [3] project was started in order to create a flexible framework that will allow different security extensions to be added to Linux. After a while, NSA started to adapt SELinux to use the LSM framework.
In general, SELinux subsystem implements a *flexible mandatory access control (MAC)* mechanism called *type enforcement (TE)*. From a puristic perspective, SELinux provides a hybrid of concepts and capabilities drawn from *mandatory access controls, mandatory integrity controls, role-based access control (RBAC)* [13], and *type enforcement* architecture. As you will see, type enforcement provides strong mandatory security in a form that is adaptable to a large variety of security goals, concurrently. Type enforcement provides a mean to control access down to the individual program level, in a manner that allows an organization to define a security policy appropriate for their systems. In TE, all *subjects* and *objects* have a *type identifier* associated with them. To access an object, the subject's type must be authorized for the object's type, regardless of the user identity of the subject. SELinux brings flexible TE along with a form of *role-based access control* and the optional addition of traditional *Multi-Level Security (MLS)* [11] [12] to Linux. This flexible and adaptable MAC security, built in to the mainstream Linux operating system, is what makes SELinux such a promising technology for improved security.

2.1 Flask Architecture

The *Flask Architecture* [2], as shown in Figure 1, describes the interactions between subsystems that enforce security policy decisions and a subsystem which makes those decisions, and the requirements on the components within each subsystem. The primary goal of the architecture is to provide flexibility in the security policy by ensuring that these subsystems always have a consistent view of policy decisions regardless of how those decisions are made or how they may change over time. Furthermore, this architecture includes an application transparency, a defense-in-depth, an ease of assurance, and a minimal performance impact.
Components which enforce security policy decisions are referred as *object managers*. Components which provide security decisions to the object managers are referred as *security servers*. The decision making subsystem may include other components such as administrative interfaces and policy databases, but the interfaces among these components are policy-dependent and are therefore not addressed by the architecture.

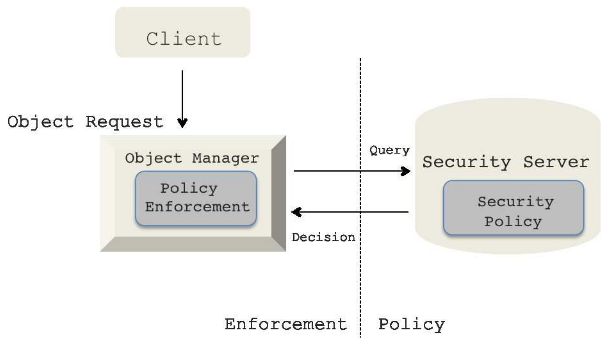

Figure 1: *The Flask Architecture*

Moreover, *flask security architecture* provides several primary elements to the object managers such as, *interfaces for retrieving access, labeling* and *polyinstantiation decisions* from a security server. The access decisions specify whether a particular permission is granted between a subject and an object, the labeling decisions specify the security attributes that should be assigned to an object, and the polyinstantiation decisions specify which member of a polyinstantiated set of resources should be accessed for a particular request. Furthermore, a vital *Access Vector Cache (AVC)* module is implemented, that allows the object manager to cache access decisions and thus minimize the performance overhead. Finally, a helpful extra ability of registration is present, in order to receive notifications of changes to the security policy. Hence, we conclude that *object managers* are responsible for defining a mechanism for assigning labels to their objects and specifying a control policy of how security decisions will be used to control the services provided. This control policy addresses threats in the most general fashion by providing the security policy with control over all services provided by the object manager and by permitting these controls to be configured, based on threat. Thus, each object manager must define handling routines which will be called in response to policy changes.

2.2 Linux Security Module (LSM)

The *Linux Security Module* (LSM) project addresses the problem of utilizing the advanced features of the newly implemented security mechanisms in the existing systems by providing the Linux kernel with a general purpose framework for access control. LSM enables loading enhanced security policies as kernel modules and aims to extend to widespread deployment of security hardened systems, by providing linux with a standard API for policy enforcement modules. Software vulnerabilities can be mitigated by effective use of access controls. Discretionary access controls (DAC) are adequate for user management, but are not sufficient to protect systems from attack. Extensive research in non-discretionary access control models has been done for over thirty years, but there has been no real consensus on which is the one true access control model. Because of this lack of consensus, there are many patches to the Linux kernel that provide enhanced access controls, but none of them is a standard part of the Linux kernel.
The Linux Security Module (LSM) project seeks to solve this Tower of Babel quandry by providing a general purpose framework for security policy modules. This allows many different access control models to be implemented as loadable kernel modules, enabling multiple threads of security policy engine development to proceed independently of the main Linux kernel. SELinux is one of the most famous enhanced access control implementations that has already been adapted to use the LSM framework. The many characteristics that LSM provides, such as the *true genericity, the ability to support the logic capabilities, the conceptual simplicity, the minimal invasiveness,* and the general *efficieciency* that delivers, seem ideal for a security subsystem such as SELinux.

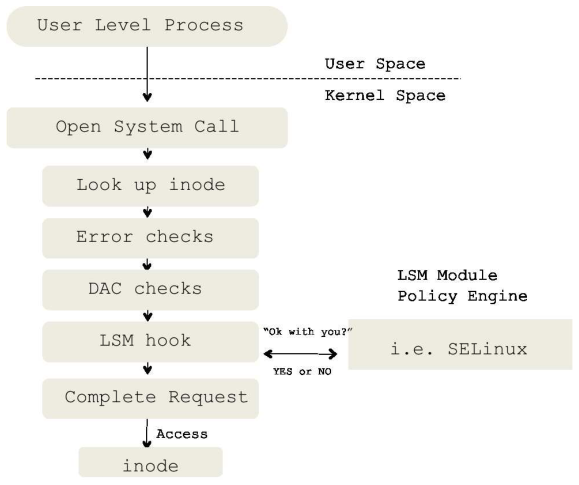

Figure 2: *The Linux Security Module*

To achieve these goals, while remaining agnostic with respect to styles of access control mediation, LSM takes the approach of mediating access to the kernel's internal objects: tasks, inodes, open files, etc. As shown in figure 2, the system call of a process-execution traverses the Linux kernel's existing logic for finding and allocating resources, performing error checking and passing the unix DAC. Afterwards, the *LSM hook* requests the permissions from the security module, (in our case SELinux) in order to allow the access to the internal object and take the security policy decision. Finally, based on the response and the retrieved access permissions from the security module and the *DAC checks*, the system will determine whether the initial request will succeed or fail.

3 SELinux Policy Language

This section presents the basic characteristics of the SELinux policy language. It tries to analyze and explain the most important policy language features, like the *Object classes and Permissions*, the *Type Enforcement* (TE) , the *Multi-Level Security (MLS)*

[7] [10] and the *Roles and Users*. Furthermore, it presents the two different methods of managing the policy language build process, *Example Policy* and *Reference Policy*.[7]

3.1 Object Classes & Permissions

SELinux, in order to apply the access enforcement mechanism in the kernel, is using object classes and their associated permissions as part of the policy language. *Object classes* are nothing more than the representation of categories, such as files, processes, directories and sockets. There is a corresponding object class for every kind of system resource. *Permissions* represent accesses to these resources, such as opening, writing, reading, executing and sending. An instance of an object class is simply called an *object*. An object class refers to the entire category of resources, in contrast to the object that refers to a specific instance of the object class. The set of permissions defined for an object class is also known as *access vector*. In SELinux, two types of permissions are defined, common permissions and class-specific permissions. *Common permissions* are a set of permissions shared by more than one object class. They are associated with the object classes as a group using the access vector. *Class-specific permissions* allow us to declare permissions specific to an object class alone. In general, an object class may have only class-specific permissions or only common permissions, and some times may have both. The access vector represents all the possible access that can be allowed to the resources represented by that object class. The set of object classes available depends always on the version of SELinux and its Linux kernel. Understanding object classes and permissions is really difficult and requires a good knowledge of both SELinux and Linux operating system.

3.2 Type Enforcement (TE)

The main part of SELinux policy is a set of statements and rules that define the type enforcement (TE) policy. TE rules express all the allowed access to resources exposed by the Linux kernel. So, an access attempt by a process to a file will succeed, only if there is at least one TE rule allowing that access. Thus, considering the number of processes and resources that a modern Linux system has, a well-defined and strict TE policy can contain thousands of TE rules. The rules are not so complex and they all fall into two basic categories, *access vector (AV)* and *type rules*. The *AV rules* are used to *allow* or *audit* access between two types and the *type rules* to control default labeling decisions. One of the most important concepts of SELinux is that TE rules associate privileges and accesses with programs and not users. In this way a program can be restricted to the minimum access permissions required to function properly, so that even if it malfunctions or has been exploited, the security of the system is not entirely compromised.The basic building blocks for TE rules, are the *types*. SELinux uses types to determine what access is allowed. Also there are some other policy features, attributes and aliases that contribute to the management and use of types. *Attributes* are used to refer to a group of types with a single identifier, and *aliases* are a convenience mechanism that allows to define alternate names for a type.

3.3 Multi-Level Security (MLS)

SELinux was modified within the definition of the Flask architecture, in order to improve MLS support and to make it more responsive to real world MLS requirements and more compatible with other MLS systems. At the time, SELinux provides optional support for *Multi-Level Security (MLS)*. Although type enforcement remains the fundamental access control mechanism of SELinux, we can also enable the optional MLS features to provide additional MLS-style mandatory access controls. MLS is another form of mandatory access control that is applicable to some security problems, especially those associated with high-classified data control. In SELinux, MLS is an optional extension to type enforcement and you cannot have MLS features without it. [11] [12]

3.4 Roles and Users

SELinux does not entirely ignore the roles and users. Roles and users exist in SELinux as the basis for its RBAC feature. It is possible for the policy to specify multiple domain types with different sets of privileges for the same program, based on the user who runs the program. Nonetheless, the level of access control is still based on the program's domain type and not the user's privileges. The security features of most other mainstream operating systems are mostly centered on granting access to users, either directly or through some form of group or role mechanism. This situation is completely different in SELinux, where access is not granted directly to users or roles. Roles act as a supporting feature to type enforcement, and together with users provide a mean to bind type-based access control with Linux users and the programs they are allowed to run. RBAC in SELinux further constrains type enforcement by defining the relationship between domain types and users to control Linux users' privileges and access permissions. RBAC does not allow access, but, as always in SELinux, allowed access is the providence of type enforcement.

3.5 SELinux Security Policies

The composition of all the elements of the SELinux policy language, in order to create a complete and comprehensive security policy, that meets all the expected security goals, can be a really difficult task for the policy writer, who tries to implement the SELinux policy language. Therefore, the methods for building security policies are rapidly changing and evolving. In SELinux, it is possible to choose between two different methods for creating and modifying the policies. This section tries to briefly describe and summarize the most important characteristics of each method.

3.5.1 Example Policy

Example Policy has been released by NSA and through the years has been evolved by the community development. It is the oldest and the most complex method for building the policy in SELinux.

The important enhancement that this method has, is the ability to build both strict and targeted policies. *Strict policy* makes maximum use of SELinux to provide seperate domain types for each program. The strict policy causes breakage with existing Linux applications, which expect looser security controls. Thus, for many users, these annoying application breakages were an unacceptable trade-off for increased security. *Target policy*, was created in order to solve the strict policy concern. It was derived from the strict example policy and aims to use SELinux to isolate high-risk system services from the rest of the system. Only the targeted services have enhanced restrictions, while all the other programs run in an unconfined domain that essentially neutralizes the enhanced security of SELinux. It is obvious, that the main difference between *strict* and *target* policies is that the target policy limits the permission sets of a few outwardly vulnerable services and provides no extra limits for local users and programs, while strict policy defines permission sets for all users and most applications and services, without distinguishing them regarding their risk levels. [7]

3.5.2 Reference Policy

The major problem of *Example Policy* is the fact that it is really difficult to understand, develop, and maintain the policy, unless you are an expert with the SELinux enforcement mechanism and the policy language. A policy writer should have detailed knowledge of the entire policy, in order to use it as the basis of new application policy modules. To overcome this problem, the Tresys Technology started a project called *Reference Policy.* This project tries to refactor the community knowledge gained through evolution of the NSA example policy, into a form that exhibits many of the strengths and features of modern software engineering, thereby making the policy more maintainable, verifiable and usable. *Reference Policy* simplifies the *Example Policy* for SELinux and is using modularity, layering, encapsulation, and abstraction. The main goal of such a reconstruction of the policy already used, is to allow greater adoption and adaption of SELinux by increasing the ability to validate the security properties of a given SELinux policy. [15] [16]

4 SELinux in a Health Information System

In a HIS there are many different kinds of applications involved, which most of the times need to interact with different access rights in several sensitive data. The *high-grained level of control*, the ability to *change the security level of blocks of data* and the ability to *change policies in abnormal circumstances* are some basic prerequisites that every security mechanism applied to a HIS must fulfill.[6] So, this section tries to describe mechanisms that SELinux provides, in order to ensure these HIS prerequisites. A source coding of the language or any implementation of those mechanisms are out of the scope of this paper, since they depend exclusively on the requirements of a specific case and they cannot be developed in a general sense.

4.1 High-grained Level of Control

SELinux policy language is capable to enforce a domain separation at the application layer, known also

as sandboxing. *Sandboxing* is efficient enough to ensure that common attacks to the system will fail or at least will be controlled and bounded. Thus, securing HIS applications inside a sandbox is possible using SELinux policy. Usually, applications such as web-healthcare-applications, which are often used in a HIS, could be controlled and restricted through SELinux Policy.
In SELinux, the policy writer is able to declare separate domains for each of the web-healthcare-applications and is also able to control the web-server (i.e. apache), the database-server (i.e. SQL) and all the other processes/daemons that interact with the web-healthcare-application. Moreover, the fact that SELinux policy language supports *conditional policies*, makes the HIS policy writer able to set conditional policy statements and enable policy rules only under specific circumstances, in which they are needed. Hence, we could say that SELinux provides a really high-grained level of control at operating-system level, as it is able to control and restrict every process running on Linux operating system, letting the HIS policy writer to modify and edit the existing policy files based on the specific HIS needs each time.

4.2 Change the Security-Level of Blocks of Data

Application data tends to be much more dynamic and flexible than data at the level of the operating-system. Since SELinux provides security mechanisms for the operation-system level, the dynamic modification of data in a higher level than this, is the most important issue that SELinux administrators have to deal with. As the change of the security level of block of data is related to whether a specific user is authorised to access these data or not, the most appropriate approach to fulfill this requirement of HIS is from the scope of users and their permissions to the database. There may be many users of an application level database, while the number of owners of operating system processes tends to be very small. By default, SELinux is configured for four users, including *system, staff, sys-admin* and *ordinary users*. Adding new users or new rules for interactions between domains and types require recompiling and reloading the configuration policy. The fact that operating-system level relationships tend to be very static, for example they should change only when new software is installed, is neither a disadvantage for the normal use cases for SELinux, nor well suited for creating rapidly changing sandboxes. Furthermore, the targeted policy does not permit application level sandboxes, because all application processes run in the unconfined domain and therefore any system supporting application level security is compelled to run in strict mode.
A solution to this complex problem, which also prevents the problem of creating additional complex interactions between application and operating system level objects, is to create a proxy. The *proxy* [17] can be viewed as a micro-instance of SELinux, that deals only with application data, and it has the following features:

- runs at the application level and is secured in its own sandbox by SELinux, preventing unwanted interactions with other processes.
- regulates access by application level processes to protected data, using its own set of configuration files.
- deals with the added levels of interaction complexity at the application layer by using an enhanced version of RBAC, in which role permissions are inherited throughout a hierarchy.

In one sense, this solution can be viewed as nested SELinux, whereby operating system level processes see only a monolithic object (the proxy) representing application processes, meaning that the number of configuration rules between the two layers is linear rather than multiplicative. Furthermore, by collating roles into hierarchy and associating the lowest member of each hierarchy with each type, the need to associate every role with every type is obviated.
As an example, a vertical slice of a role hierarchy may consist of "Doctor" is a subset of role "Clinician" and "Surgeon" is a subset of role "Doctor". Configuring the policy with any user in the role of "Clinician" has access to *type y* automatically covers the rules for any user in the role of "Doctor" has access to *type y* and any user in the role of "Surgeon" has access to *type y* by virtue of their membership of the family. Portions of the hierarchy can be overridden: configuring any user in the role of "Surgeon" not to have access to *type y*, does not cause a contradiction, but allows only "Clinicians" and "Doctors" to access *type y*.
Actually, SELinux uses a primitive version of RBAC and supports a variant of $RBAC_2$ to associate individual users with specific types, and to disassociate individuals operating in different roles. The standard form of the domain and object contexts are *User with Role owning Type*.

This allows the system to differentiate between *Doctor W* accessing the internet using a web-browser, and the same *Doctor W* using a medical application. In the former case, sensitive medical records pertaining to *Doctor W's patients* should be inaccessible. So although the same user is involved, the combination of user and roles, in which he or she is involved, is not. Again, RBAC is an appropriate mechanism to secure this kind of data. However, the level of granularity of SELinux is high, but medical databases support records have varying privileges, which have a much finer level of granularity. Consequently, it is also responsibility of the medical application to ensure that its records are not accessed by clinicians with inappropriate levels of authority. Role hierarchies have been previously proven useful in flexibly defining record access in medical scenarios, but are unsupported in SELinux.

4.3 Change Policies in Abnormal Circumstances

One of the properties, the SELinux policy should satisfy, in order to be adequate for security of HIS, is the ability to change policies in abnormal circumstances. The ways, in which this requirement can fulfilled is descripted in this section.

There are some cases, when records must be accessible even in the absence of legitime credentials. For example, in cases that the patient is unconcious or if the authorized viewer of a patient's case is not present, but the patient needs emergency treatment, then the availability of the information is more important than its privacy. In both cases, there would be no time to rewrite the security policies, but the ability to switch to another policy set to suit the emergency scenario would be beneficial. This could be facilitated by a *Risk Assessment Unit*, shown in figure 3, which is a unit that evaluates the risk and the emergency of a situation. When the data are proved sufficient to change to emergency mode, then irrespective of the level of security, all underlying activities are audited.

If the HIS includes a proxy mechanism, which we described in the previous section, it is also possible to audit the medical records in abnormal circumstances. The proxy is programmed to respond to a special role of *Emergency*, in which case it moves into auditing mode, until a new set of credentials with a differing role is provided. In auditing mode, all records can be accessed and modified, but each action is recorded for review by the security administrator, so that in-

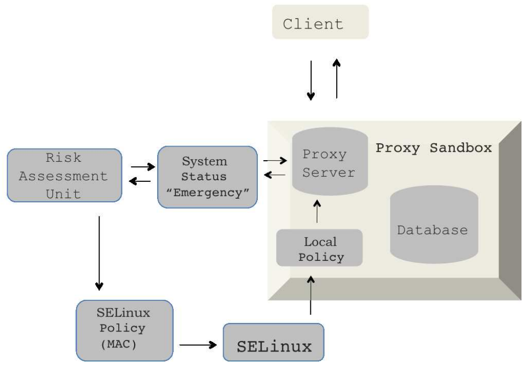

Figure 3: *Risk Assessment Unit*

appropriate use of these data could be recognised and punished.

4.4 Can SELinux be trusted in a HIS

In the subsection 4.2 we described in which extend the implementation of SELinux in a HIS can fulfill its three basic requirements. At this point, we will examine the effectiveness and the weaknesses, that must be considered, before applying SELinux into a HIS.

Firstly, we could say that the fine-grained flexibility provided by SELinux at the level of files and sockets and its architectural modularity seem to offer a high-grained level of control and a high level security in application data. In this sense, the security of sensitive data, such as the health records, is guarantered. On the other hand, the fact that the SELinux offers a fine-grained flexibility can also be a disadvantage. In order to provide this feature, a configuration file of about 50.000 lines long must be dynamically maintained. Therefore, reconfiguring the policy or solving problems caused by bugs can be a difficult task for the SELinux administrator. Considering also the fact that SELinux is an open source project and is available for different Linux OS distributions (Fedora, Red Hat, Debian, etc.), that maintain their own configuration file, this task becomes even more tedious.

Secondly, when it comes to the need of changing the security level of blocks of data, SELinux seems to be inappropriate. Although this feature is provided by SELinux using $RBAC_2$, it is difficult to be configured. As it is mentioned in 4.2, SELinux uses only a form of RBAC model, but it is limited supported and complex at the time. Furthermore, the need of recompiling after adding new roles & users is a seri-

ous impediment to the success of an enterprise system such as HIS.
Finally, looking at how SELinux handles the change of policies in abnormal circumstances, we could say that is capable enough to support and monitor them, through a *Risk Assessment Unit* which facilitates this whole procedure. Just an override function has to be accomodated in SELinux, so that an unauthorized user gains temporary access in a health record, under situations of emergency.
Taking a general glance into SELinux, it is clear that the advantages and disadvantages of this security subsystem can be expressed in terms of the purpose and the security requirements of the system, in which it is implemented. Nevertheless, there are some general advantages and disadvantages, that come from the implementation of SELinux. The main advantage of SELinux is the *seperation between policy logic and security enforcement*, which improves expressiveness and effectiveness in decision making. But SELinux is usually associated with inconvenience security and lost of productivity, due to the difficulty of reconfiguring the SELinux policy language. Of course, this last disadvantage of SELinux, does not mean that it can not be trusted for a HIS, it means only that it is a complex subsystem on Linux OS that needs a special and careful administration.

5 Conclusions

This paper tried to analyze and decide whether the SELinux subsystem could be efficient enough to secure the sensitive data of a Health Information System (HIS). As already mentioned, securing efficiently a HIS Infrastructure involves the effort to secure all levels of this system, such as *application level, operating system level, hardware level* and thus a separate approach must be made, concerning security for each of these levels. This paper focused on the operating system level and specifically on the SELinux subsystem, which could be consisted as a module on a Linux distribution through Linux Security Module. It is concluded that, despite of the security flexibility and the detachment of the policy enforcement from the policy configuration logic that SELinux provides, it is still too complex and difficult in implementation. A HIS administrator must be as sure as possible for the security of its system and must be able to maintain the security easily and dynamically without much effort. Without a doubt, the granularity of SELinux is sufficient to elegantly secure application data, but this is not enough and unfortunately further mechanisms are required in the application layer. SELinux, should provide support for $RBAC_2$ in a different form, allowing inherited permissions and simplifying SELinuxs notoriously complex configuration process. Furthermore, it should have a graceful mechanism for handling changes within policies, in order to administrate highly dynamic environment, such as a HIS. Also, SELinux must definitely provide a satisfactory mechanism to change policies when circumstances change, without requiring strong auditing actions. At this time, SELinux seems a very promising security system, but more research needs to be done, in order to improve and facilitate the way SELinux can be implemented in systems in the extended enterprise. Indeed, there are some efforts made in this direction by Tresys Technology [15], that tries to implement the Reference Policy in SELinux, in order to reconstruct in a simplified way the SELinux policy already used. As SELinux becomes more popular, it gains positive treatment from the public community and the enterprise industry, and therefore it is expected to be more trusted and efficiently implemented in HIS in the future.

References

[1] *Peter G.Goldschmidt,* (2005), "HIT & MIS: Implications of Health Information Technology and Medical Information Systems".

[2] *Michael Hafner, Mukhtiar Memon and Muhammad Alam,* (2008), "Modeling and Enforcing Advanced Access Control. Policies in Healthcare Systems with SECTET", University of Innsbruck, Austria

[3] *GAO,* (2006), "Information Security: Department of Health and Human Services Needs to Fully Implement Its Program", United States Government Accountability Office.

[4] *OpenClinical - Knowledge Management for Medical Care,* "Health Information Technology adoption, programmes and plans: Europe (European Union)". ,(Accessed: 22/11/2008), `http://www.openclinical.org/hitGlobalEuropeEU.html`

[5] *Vicky Liu, Lauren May, William Caelli and Peter Croll,* (2007), "A Sustainable Approach to Security and Privacy in Health Information Systems", Queensland University of Technology.

[6] *Matt Henricksen, William Caelli, Peter Croll,* (2007), "Security Grid Data Mandatory Access Control",Information Security Institute, Queensland University of Technology.

[7] *Frank Mayer,, Karl MacMillan,, David Cap,* (2006), "SELinux by Example: Using Security Enhanced Linux ", Prentice Hall.

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

[9] *Chris Wright, Crispin Cowan, James Morris, Stephen Smalley, Greg Kroah-Hartman,* (2002), "Linux Security Module Framework", WireX Communications, Inc., Intercode Pty Ltd, NAI Labs, Network Associates, Inc., IBM Linux Technology Center.

[10] *Stephen Smalley,* (2002), "Configuring the SELinux Policy", Trusted Computer Solutions Inc.

[11] *D.E.Bell and L.J.La Padula,* (May 1973), "Secure Computer System: Mathematical Foundations and Model", Technical Report M74-244,The MITRE Corporation, Bedford.

[12] *Chad Hanson,* (2006), "SELinux and MLS: Putting the Pieces Together", Trusted Computer Solutions Inc.

[13] *D.F.Ferraiolo, J.A.Cugini and D.R.Kuhn,* (1995), "Role-Based Access Control (RBAC): Features and Motivations", In Proceedings of the 11th Annual Computer Security Applications Conference.

[14] *Bill McCarty,* (2004), "SELinux NSA's Open Source Security Enhanced Linux", O'Reilly.

[15] *Tresys Technology - Open Source Software,* "Security Enhanced Linux Reference Policy". ,(Accessed: 25/11/2008), `http://oss.tresys.com/docs/refpolicy/api/`

[16] *Christopher J.PeBenito, Frank Mayer, Karl MacMillan,* (2006), "Reference Policy for Security Enhanced Linux", Tresys Technology.

[17] *Peter R.Croll, Matt Henricksen, Bill Caelli and Vicky Liu,* (2007), "Utilizing SE Linux to Mandate Ultra-secure Access Control of Medical Records", Information Security Institute, Queensland University of Technology, Brisbane, Australia.