LB/DON/63/2020

DCS 05/123

GENERIC SELINUX RULES & POLICIES FOR SECURE EXECUTION OF NETWORK SERVICES IN LINUX

Mario Roshane Ishara Fernando

158214U

LIBRARY
UNIVERSITY OF MORATUWA, SRI LANKA
MORATUWA

Dissertation submitted in partial fulfillment of the requirement for the degree Master of Science in Computer Science

Department of Computer Science & Engineering

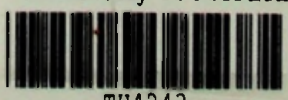
University of Moratuwa

November 2018

004 "16"
004(043)

University of Moratuwa
TH4243

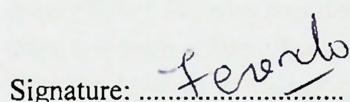
TH 4243 + CD-ROM
is not working



DECLARATION

I declare that this is my own work and this dissertation does not incorporate without acknowledgement any material previously submitted for degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

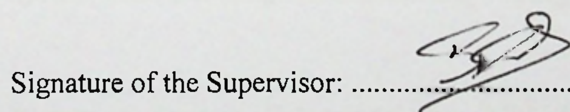
Also, I hereby grant to University of Moratuwa the non - exclusive right to reproduce and distribute my dissertation, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works.



Signature: .... Date: 16/11/2018

Name: M.R.I. Fernando

I certify that the declaration above by the candidate is true to the best of my knowledge and that this report is acceptable for evaluation for the Masters of Dissertation.



Signature of the Supervisor: .... Date: 16/11/2018

Name: Dr. Shantha Fernando

Signature of the Co-Supervisor: .... Date: ....

Name: Dr. Chandana Gamage

ABSTRACT

Usage of Network services and network stack-based applications on Linux systems are increasing rapidly, hackers around the world exploit security flaws there by executing sophisticated attacks on these services and compromising the entire system. Applying SELinux policies to a system which serves multiple network services has been a challenge due to policy conflicts. These policy conflicts are overridden by the security administrator there by applying SELinux rules to make the network services operational, however this might result in loop holes thereby information leakage from one or multiple services to another. This results in compromisal of not only the network service being attacked but other running services in the system which might lead to the entire trusted computing base being compromised. Deployment of SELinux Multi Level Security mandatory access control is an appropriate model to be applied over a system where we can segregate information flow from various security levels into the level of even categorized compartments. However, when running multiple network services over a single SELinux MLS enabled system, it is required to determine the security levels to be labelled over the subjects and the objects of the respective network services to overcome the ambiguity of the security levels in the information flow of a security lattice. Preserving both confidentiality and integrity of a system is a challenge and it is required to find the most secure way of information flow in a security lattice while achieving it using the existing SELinux MLS framework. This research focuses on a number of access control models, security models, lattice-based access control models and a wide range of SELinux security policy implementations. The goal of this research is to determine the security labels and security levels of the network services intended to run on a SELinux MLS enabled system while allowing information flow through the security lattice only if required.

Keywords: Security Enhanced Linux, Multi-Level Security, Bell Lapadula Model, Mandatory Access Control, Security Lattice, Type Enforcement, Sensitivities, Categories, Security Contexts

ACKNOWLEDGEMENT

I would like to express my special thanks of gratitude to my project supervisors, Dr. Shantha Fernando and Dr. Chandana Gamage for their patient guidance, enthusiastic encouragement and useful critique of my research work. Their willingness to give their time so generously have been very much appreciated. Especially mentioning the moral support and the continuous guidance by providing important feedback enabled me to complete my work successfully.

Also, I would like to thank Dr. Stephen Smalley from National Security Agency, USA and Mr. Dan Walsh from RedHat Incoperation, USA who supported me to carry out this research by providing valuable resources regarding the SELinux Multilevel Security domain. Finally, I would like to thank my spouse who supported and encouraged me throughout this project.

TABLE OF CONTENTS

DECLARATION i
ABSTRACT ii
ACKNOWLEDGEMENT iii
TABLE OF CONTENTS iv
LIST OF TABLES viii
LIST OF FIGURES ix
LIST OF ABBREVIATIONS x
1. INTRODUCTION 1
1.1. Background 1
1.1.1. Securing Network Services 1
1.1.2. Security Model of SELinux 2
1.1.3. Mandatory Access Control Model 2
1.1.4. Security Enhanced Linux and the Linux kernel 2
1.2. Problems with SELinux & running multiple network services 3
1.3. Research Problem 3
1.4. Research Objectives 4
1.5. Proposed Solution 4
1.6. Research Methodology 5
1.7. Organization of the Thesis 6
2. LITERATURE REVIEW 7
2.1. Overview 7
2.2. Access Control mechanisms and Security Models 7
2.2.1. DAC and MAC 7
2.2.2. Multi Level Security 9
2.2.3. Security Models 9
2.2.4. Bell Lapadula Model 10
2.2.5. Problems with Security Models 12
2.3. Security Objectives (Confidentiality and Integrity) 13
2.4. Lattice Based Access Control 13
2.4.1. Orders and Lattice 13
2.4.2. Lattice for Classifications 14

2.4.3. Finite Lattice ... 14

2.5. A Security Lattice for MLS ... 15

2.5.1. Information flow in a lattice with Multi Level Security ... 16

2.6. SELinux Overview ... 17

2.6.1. Introduction ... 17

2.6.2. Subjects and Objects ... 18

2.6.3. Flask Architecture ... 19

2.6.4. Type Enforcement Mechanism ... 21

2.6.5. Security Context ... 21

2.6.6. Types and Attributes ... 21

2.6.7. Access Vector Rules ... 22

2.6.8. Domain Transition ... 22

2.6.9. Constraints ... 23

2.6.10. SELinux Policies ... 27

2.6.11. Policy Modules ... 28

2.6.12. Summary of SELinux Overview ... 29

2.7. SELinux Multi-Level Security ... 29

2.7.1. Security Levels ... 30

2.7.2. Security Level Translation ... 32

2.7.3. mlsconstrain Statements ... 33

2.7.4. mlsvalidatetrans statements ... 35

2.7.5. Privilege Management ... 36

2.7.6. Complexitiy of SELinux policies ... 37

2.7.7. SELinux policy analysis tools ... 37

2.7.8. Problems with SELinux analysis tools ... 37

2.7.9. SELinux Challenges and Proposed Solution ... 38

3. METHODOLOGY ... 39

3.1. Lattice labels for SELinux ... 39

3.2. Example of a dominance relationship ... 41

3.3 Bell Lapadula Model review ... 42

3.4 SELinux Policy alignment with Bell Lapadula model ... 43

3.4.1 SELinux Policies facts in Customized Bell-Lapadula model ... 44

3.5 Overriding Bell Lapadula custom policy ... 44

3.5.1 Bell Lapadula Simple property in SELinux mlsconstrain statements 44
3.5.2 Bell Lapadula custom *-property in SELinux mlsconstrain statements.... 46
3.6 SELinux policy Type enforcement and MLSConstrain evaluation order........ 48
3.7 Security Lattice for SELinux Type Enforcement & SELinux MLS 49
An environment with only Type Enforcement (No Multi Level Security) 49
An environment with SELinux Type Enforcement + Multi Level Security where: 50
An environment with special type mapped to an attribute in MLSConstrain statements 51
3.8 Type Enforcement vs MLSConstrain vs Bypassing MLSConstrains 52
3.9 Program to determine the Security levels for secure execution 53
3.9.1 Decision making on determining security labels........ 54
3.9.2 Program Workflow 57
3.10. Assigning the Security labels for secure execution 63
3.11 Generic SElinux policies for secure execution 64
4. SYSTEM/SOLUTION ARCHITECTURE AND IMPLEMENTATION 66
4.1. Interactive program to determine the security levels 66
4.1.1. Pseudocode for the program 67
4.2. Setting up SELinux MLS testbeds 68
4.3. Running the Program 70
Scenario 1 70
Scenario 2 71
4.4. Choosing the network services........ 72
4.4.1. SELinux module creation for Scenario........ 72
4.4.2. SELinux module creation for Scenario 2........ 78
5. System Evaluation and Analysis 82
5.1 Analysis of testbed 1 environment........ 82
5.1.1 State Diagram for Testbed1 84
5.2 Analysis of testbed 2 environment........ 84
5.2.1 State Diagram for Testbed2 environment........ 85
5.2.2 Problem Analysis in Testbed2 environmet........ 85
5.3 Verifying the generic SELinux Rules 89
6. CONCLUSION 91

REFERENCES 93

APPENDICES 97

Appendix A – Comparison of SELinux Analysis tools 97

Appendix B – State Diagram for testbed1 environment how information flow occurs for wordpess application to function 98

Appendix C – State Diagram for testbed2 environment how information flow occurs for wordpess application malfunctions 99

LIST OF TABLES

Table 2.1 Comparison of Access Control in Standard Linux and SELinux 17
Table 2.2 MLS Constraint Table.. 34
Table 2.3 MLSConstraint Conditional Operators.. 35
Table 2.4 SELinux Attributes 36
Table 3.1 Permutations of how the final decision is made upon the allowance and denials of Type Enforcement Rules, MLSConstrain rules and bypassing MLSConstrain Simple property / Custom *-Property rules...................................... 53
Table 3.2 Inequality Table utilized by the Python script 57
Table 3.3 SELinux Generic Rule Precedence.. 65
Table 4.1 Filesystem file relabelling of a SELinux MLS enabled system................. 69

LIST OF FIGURES

Figure 2.1 Trojan horse Example 8
Figure 2.2 Multi Level Security (MLS) 9
Figure 2.3 Available data flows using an MLS system 11
Figure 2.4 Equivalance of BLP and BIBA 12
Figure 2.5 Lattice Information Flow 14
Figure 2.6 MLS Security Lattice 15
Figure 2.7 The security lattice in Virtualvault 16
Figure 2.8 High Level SELinux Components 18
Figure 2.9 Flask Architecture 20
Figure 2.10 Domain Transition 22
Figure 2.11 Conceptual Diagram of RBAC 23
Figure 2.12 Bell Lapadula Model 30
Figure 2.13 App Program Security Policy Rules in SELinux 38
Figure 3.1 Bell-Lapadula Security Model 42
Figure 3.2 Modified Bell-Lapadula Model 43
Figure 3.3 Lattice information flow for type enforcement rules 50
Figure 3.4 Lattice Information flow for Services running in labels S0.C0.C3 and S1.C0.C1 51
Figure 3.5 Unrestricted Information Flow by bypassing MLSConstrain rules for Simple and custom *-Property 52
Figure 3.6 0000 58
Figure 3.7 0001 58
Figure 3.8 0010 58
Figure 3.9 0011 59
Figure 3.10 0111 59
Figure 3.11 1000 59
Figure 3.12 1001 60
Figure 3.13 1010 60
Figure 3.14 1011 61
Figure 3.15 1111 62
Figure 5.1 Wordpress is up and running 84
Figure 5.2 Wordpress is not functional as apache process is unable to write to mysql socket file 84
Figure 5.3 Wordpress is not functional as apache process is unable to write to mysql socket file 85

LIST OF ABBREVIATIONS

BLP	Bell-Lapadula Model
DAC	Discretionary Access Control
GNU/GPL	GNU's NOT UNIX / General Public License
LSM	Linux Security Module
MAC	Mandatory Access Control
MLS	Multi Level Security
NSA	National Security Agency
RBAC	Role Based Access Control
SELinux	Security Enhanced Linux
TCB	Trusted Computing Base
TE	Type Enforcement

1. INTRODUCTION

1.1. Background

SELinux is a Linux kernel security module which facilitates various mechanisms & procedures to control security policies by modification of the Linux kernel and user-space tools that is supported by any Linux distribution. The concepts were initially developed by the National Security Agency (NSA), United States which is open source under GNU/GPL License in year 2000 [1] [2]. It follows the MAC model where all permissions are denied by default and any permission operation should be permitted explicitly which provides an enhanced mechanism to vastly improve information confidentiality and information integrity in a lattice-based access control model. SELinux has several security policies developed which could be applied over a linux system which is based on Type enforcement, role-based access control (RBAC) and multilevel security (MLS) [3].

1.1.1. Securing Network Services

Majority of modern attack vectors occur due to systems that are exposed to the outside world which could run insecure, buggy or vulnerable network services. Also, it should be taken into the consideration that for a system to run multiple network services, the system is at higher risk and at a higher probability of getting attacked than a system which runs just a single service. The reason behind this is a single service running on a system can be tightened with many security policies to minimize any security flaws whereas for a system running multiple network services the probability of an attack would be relatively high due to many reasons. One thing could be due to network service specific vulnerabilities or unprotected information flowing in and out between network services unless really required. Though the SELinux MAC policy rules pertaining to a corresponding network service provides fine grained security still it might weaken the security of the remaining services running on the system which has the possiblitiy to change the trustworthiness or integrity of the rest of the services running in the system due to inappropriate allowed information flows. Thus, it is quite a tough and a hectic job to maintain security of all network services on such systems [4] to make sure no unwanted information can flow through the security lattice.

1.1.2. Security Model of SELinux

Security models are often regarded as a formal presentation of the security policy enforced by the system [NCSC 1988] and are used to test a policy for completeness and consistency [5]. They describe what mechanisms are necessary to implement a security policy [6]. Over the years, many security models have been developed depending on the requirements of security objectives such as to preserve confidentiality alone, integrity alone or to preserve both confidentiality and integrity. For an example, Biba model was developed intending to preserve the integrity of information [7], Bell and Lapadula developed a model to preserve the confidentiality of the system. SELinux security policies is a modified version of the Bell Lapadula model intending to achieve such security objectives [7].

1.1.3. Mandatory Access Control Model

Mandatory access control, which, according to the United States Department of Defense Trusted Computer System Evaluation Criteria is "a means of restricting access to objects based on the sensitivity (as represented by a label) of the information contained in the objects and the formal authorization (e.g. clearance) of subjects to access information of such sensitivity" [7]. Mandatory access control restricts the access of subjects to objects on a system-wide policy and denies full access and control to users even if they created the objects (Where as in DAC this is not the case).

1.1.4. Security Enhanced Linux and the Linux kernel

Security-Enhanced Linux (SELinux) is a Linux kernel security module that provides a mechanism for supporting access control security policies, including United States Department of Defense–style mandatory access controls (MAC). SELinux is a set of kernel modifications and user-space tools that have been added to various Linux distributions. Its architecture strives to separate enforcement of security decisions from the security policy itself which is also refered to as the flask architecture and streamlines the amount of software involved with security policy enforcement. The key concepts underlying SELinux can be traced to several earlier projects by the United States National Security Agency (NSA) [8].

1.2. Problems with SELinux & running multiple network services

Due to the complex nature of SELinux type enforcement rule sets and its policies [1], many Linux based systems run with SELinux disabled mode. Even if SELinux is in enabled or in permissive mode, the SELinux rule sets and the policies for a given set of the network services running on a single Linux system might not be properly applied for secure information flow in a lattice-based model. This might result in security flaws and vulnerabilities hence the probability of modern attack vectors being executed on such systems increases at higher rates resulting in compromising of many Applications and Systems which runs on Linux based distributions [5] due to unwanted information flow through the security lattice. Choosing and deciding the correct security labels, sensitivity and category levels to run the network services in the most secure manner is ambigious and this is based upon how information is shared and modified among the services which needs to interact with each other.

1.3. Research Problem

SELinux already has type enforcement mandatory based access control approach where the Security administrator has the option of enabling Multi Level security to apply on all the subjects and objects of the linux system achieved through labelling [9]. Decisions of information flow between subjects and objects are decided by the SELinux security policy server enforced by the SELinux enforcement server, which is the Linux kernel. However, when running multiple network services there should be a mechanism for the Security Administrator to decide what exact labels, sensitivity levels and optionally any categories which should be applied over the subjects and the objects of the respective network services. That is, the Security administrator can run the network services in different security levels or in the same security level depending on the information flow paths of the network services running on the single system. Therefore, there should be a mechanism to determine & thus overcome the ambiguity of security levels to preserve both confidentiality and integrity using Lattice information flow approach.

1.4. Research Objectives

In achieving objectives of security, it is vital to minimize privileges and hence to attain least privileges while allowing the systems or applications to perform its normal operations. Compromisal of one or more network services might result in compromising the entire Linux System resulting in catastrophic conditions. So therefore, even if a network service is compromised then still the remaining services shouldn't be compromised and then it's a matter of recovering the hacked service to bring up the linux system into a fully operational state if possible. A security administrator should be able to decide the exact security levels to run the network services intending to serve on a linux system. Hence a given network service will always attain the least privileges and only the required amount of information will flow for the service to be operational and serve its intented purposes. Thus, achieving the minimal privileges approach for network services achieved through SELinux Multi Level security can be the solution to overcome the entire system not being hacked due to the compromisal of one or more network services but only the attacked corresponding network service will be affected. Therefore, the approach is to determine the different security levels to overcome the ambiguity of security levels in a lattice information flow model to preserve both confidentiality and integrity for running multiple network services in a SELinux MLS enabled system.

1.5. Proposed Solution

A security administrator should have the knowledge of information flow and any interactions which occurs between the network services inorder to determine and assign the most secure SELinux labels for the network services. The combination of information flows between the respective services can be modelled into a security lattice on how information flows between the services. The security labels of the respective services are determined using an inequality form to overcome the ambiguity of sensitivity levels of the respective services. Once these sensitivity levels are determined, the corresponding sensitivity levels can be applied over the required subjects and the objects of the network services intending to run on the SELinux MLS enabled system. A generic rule procedure is presented to be followed by the Security Administrator to securely execute the network services to align with the information flow requirements to and from the services.

1.6. Research Methodology

Running multiple network services on a linux system in a Selinux enabled environment can be problematic if proper security labels aren't applied on the components of the respective network services.

There is a high chance when one service is compromised this could affect and result the compromisation of other running services and may be the entire system.

To understand the problem in a more detailed manner, the concept of mandatory access control, security models, type enforcement, multi level security was reviewed and the practical implementations such as SELinux type enforcement, SELinux Multi category security and SELinux MLS was studied using various literature.

The security models and concepts utilized for the SELinux model were studied inorder to refine the existing problems with properly applying SELinux multi level security on a given linux system.

To narrow down the problems with current SELinux implementations, communication happened with various researchers of SELinux in the National Security Agency and Redhat using various mailing lists.

The practical implementation of running SELinux with various models such as running only with Type enforcement, Multi Category Security and Multi Level security was tested against various scenarios of test bed implementations on locally set upped environments.

To understand how SELinux policy enforcement works, common network service stacks such as running apache coupled together with mysql was chosen to run against SELinux type enforcement environments and SELinux MLS enabled environments

Analysis of information flow between the subjects and the objects of apache and mysql were studied using various SELinux policy analysis tools.

A labelling structure for the network services intending to be run on the linux system was developed to determine the appropriate security levels for the respective services.

A generic rule procedure is presented so that the procedure can be followed depending on the information flows required between the network services.

The determined security labels were then applied against two network services in an SELinux MLS enabled environment two testbeds.

Functionality tests were done against the testbeds which runs the network services by verifying information flows using an application installed in the respective testbed environments.

1.7. Organization of the Thesis

Chapter 2 covers various access control models, comparison of security models, goals of information security, how the mandatory access control model is ported into the Linux kernel,fundamental concepts of SELinux policy and flask architecture, SELinux MAC model, SELinux policy language using macros, SELinux type enforcement, SELinux MLS approach,the challenges encountered due to the complexity of SELinux with proposed solutions and a comparison of SELinux policy analysis tools.

Chapter 3 covers how the security lattice-based approach & the customized Bell-Lapadula security model to study information flow could be utilized to run two network services thereby to determine and overcome the ambiguity of security levels of the lattice depending on how information should flow and modified between the 2 services. The SELinux security server precedence of logic on how type enforcement, constraints ans how these constraints can be bypassed to override the bell lapadula security model is discussed.A program needs to be designed such that it can determine the security levels of the 2 services considering the information flows which needs to occur using the Lattice based information flow approach. A table of various combinations of access rights required for the 2 services are put up to build up the logic of the script.Finally a table has been included which is a generic rule procedure to be chosen by the the Security Administrator inorder to assign the appropriate labels to secure the information flow in the security lattice considering the combination of type enforcement, constraint rules and to restrict bypassing the bell lapadula security policy using the output of the script.

Chapter 4 Two identical testbed environments SELinux MLS enabled environment is set upped so that the 2 selected services will run on the security levels determined by the python script.The script is interactive and prompts the Security Administrator on various information flows which could occur between the two services and finally the script will produce an inequality of the sensitivity levels which should be assigned on the two services intended to run on the SELinux MLS enabled system. Apache and mysql was chosen for the two network services which is installed in the respective testbeds. The two testbeds are to be utilized for two chosen scenarios as indicated in the generic rule procedure discussed in Chapter 3.

Chapter 5 An analysis is done on the respective testbeds using a state diagram & to demonstrate on how information flow is allowed and denied between the two services based on the generic rule procedure discussed in Chapter 3 for the two scenarios. Wordpress application is installed in the respective testbeds to demonstrate how information flow restrictions affects the operationability of the wordpress application.

Chapter 6 Concludes the thesis by discussing on how the generic rule procedure can be followed and used to restrict any unrequired information flow in addition to the mandatory access controls already enforced by the existing SELinux policies.

2. LITERATURE REVIEW

2.1. Overview

To understand the notion of security for information systems, access control mechanisms and real-world application of security models was reviewed along with their intentions and weaknesses with comparison. Goals of information security include mainly preserving confidentiality and integrity of data & information. For a Linux system this is achieved using Linux Security Module (LSM) ported into the Linux kernel to name it as Security Enhanced Linux (SELinux) thus forming a Trusted Computing Base (TCB) [3]. SELinux supports mainly two kinds of Policies (Targeted and MLS) [10], the complex model of SELinux policies [1] make it difficult to understand the information flow within a Linux system. Thus analyzing & understanding the information flow within a Linux system starting from the booting phase & how the system interacts with network services is the main goal to achieve a custom secure robust set of SELinux policies to minimize modern attack vectors and to prevent information leak through vulnerable or any compromised network service(s) [11].

2.2. Access Control mechanisms and Security Models

2.2.1. DAC and MAC

Access control is typically defined in one of two ways, either discretionary or mandatory access control. First, discretionary access control (DAC) is user-based. DAC gives ownership to the objects in the system. The owners of objects can in turn give access to others to use these objects. This model allows the most flexibility but is the most hectic to maintain [7]. In the second hand, in mandatory access control (MAC), objects are given a classification level and each user in the system is mapped with a clearance level. The users are then allowed to view objects based on their mapped clearance level [7].

MAC vs DAC

There is a fundamental difference between DAC and MAC which are:

1. Unrestricted DAC allows information from an object which can be read to any other object which can be written by a subject.
2. The need for additional low-level security (Kernel Level) such as the Role of the User, trustworthiness of a program, and the function of the program or the sensitivity and Integrity of the data [9] is not addressed by DAC.
3. User identity & ownership of files is the basis of Discretionary Access Control (DAC) & whereas Mandatory Access Control (MAC) is a necessary control which facilitates strong separation of applications that permits the safe execution of untrustworthy applications [9].
4. Denies full access and control to users even if they created the objects (Where as in DAC this is not the case).
5. The system security policy set by the Security Administrator determines the access writes granted.
6. MAC requires all those who create, access and maintain information to follow rules set by administrator [12].
7. Protection against malicious code is impossible using DAC since the DAC mechanism is such that every program executed by that user inherits all of the privileges associated with that user [9] as shown in Figure 2.1.

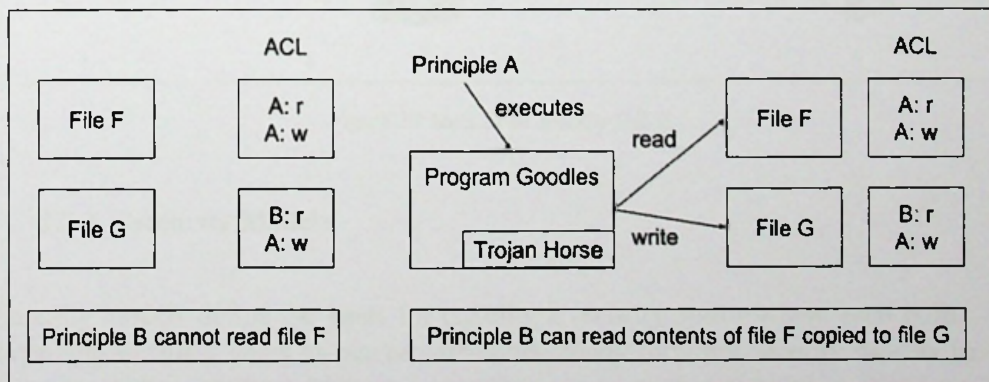


Figure 2.1 Trojan horse Example

2.2.2. Multi Level Security

Security goal of MLS is confidentiality where it intends to protect secrets from leaking between different subjects.

MLS facilitates the capability of a computer system to carry information with different levels of sensitivities, permit simultaneous access by users with different levels of security clearances as described in Figure 2.2. Thus, to prevent any user from obtaining access to information for which they lack authorization or has no permissions.

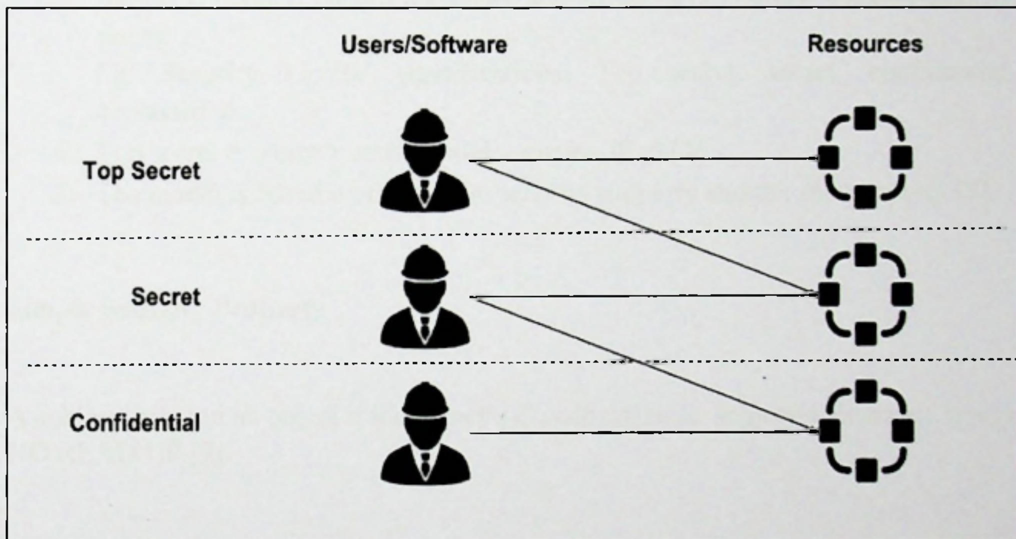


Figure 2.2 Multi Level Security (MLS)

2.2.3. Security Models

Security models define the basis for creating a security policy where each policy is intending to fulfill goals as per requirements. Many different security models have been created to address different security concerns as the security requirement of organizations could differ according to the nature of the data and the users interacting with the information. The goal of all security models is to define the authorized and unauthorized states of a system and to restrict the system to moving into an unauthorized state. The models can be either based on mandatory or discretionary access control where security could be enforced through a software independent conceptual model. Security models have been developed based on the type of system that they will be used on such as to provide security for operating systems [13].

2.2.4. Bell Lapadula Model

A military style model which does not allow leakage of information to those who are not allowed to access this information unless explicitly specified [7]. This model addresses on information confidentiality and ensures that a computer can securely process classified information which has the following characteristics:

1. Basically, information should NOT flow downwards.
2. The model however deals with confidentiality and not Integrity of information.
3. A computer system is modeled as a state transition system, in each state, each subject is given a level (clearance), and each object is given a level (security class).
   Eg: Security Levels/ classifications: Top secret, secret, confidential, unclassified.
4. Top secret > secret > confidential > unclassified [7].
5. The model is based on the simple security property and the star property [7].

Simple Security Property

A subject can read an object if the object's classification is: subject's clearance level - NO READ UP [7].

Star Property

A subject can write to an object if the subject's clearance level is: object's classification level - NO WRITE DOWN [7]. As shown in Figure 2.3, a low-level subject can write to an object with a higher classified level, therefore BLP strictly focuses on confidentiality and not on Integrity of data [7].

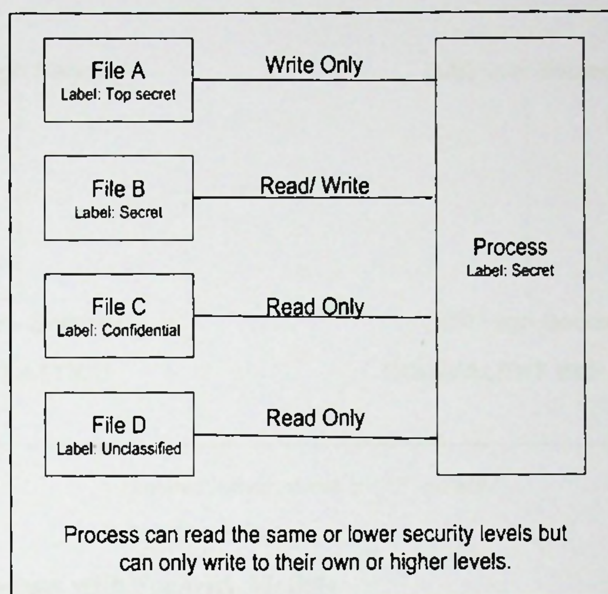


Figure 2.3 Available data flows using an MLS system

BLP and BIBA comparison

BLP and Biba are fundamentally equivalent and interchangeable [14] but however there is a significant amount of differences between these 2 models which are:

1. Lattice-based access control is a mechanism for enforcing one-way information flow, which can be applied to confidentiality or integrity goals [24].
2. BLP formulation with high confidentiality is at the top of the lattice, and high integrity at the bottom [24].
3. Information flow in the Biba model is from top to bottom & whereas in BLP model it's from bottom to top.
4. Since top and bottom are relative terms, the two models are fundamentally equivalent as indicated in Figure 2.4.

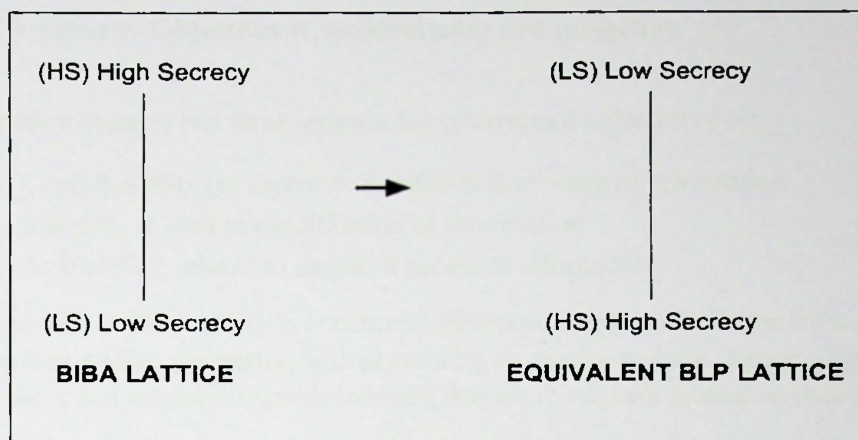


Figure 2.4 Equivalance of BLP and BIBA

2.2.5. Problems with Security Models

Even when a security model proves that a system is secure, the system may be vulnerable. The human element and social engineering have allowed even the most secure systems to be compromised. Also, security mechanisms that are at one time thought to be secure in the future can be found to be insecure.

1. Security models have theoretical limits and it is impossible to always prove that a model satisfies certain security conditions.
2. Security models based on strict mathematical properties can lead to systems that are totally unusable.
3. Building systems from rigorous mathematical security models is extremely time consuming and costly and majority of commercial systems will not be based on formal models.
4. Security models and formal methods do not establish security.
5. In other words, systems are hacked outside the model's assumptions.
6. Provable security, even if it were achievable, is not a panacea [13].

2.3. Security Objectives (Confidentiality and Integrity)

Information security has three separate but interrelated objectives [14].

1. Confidentiality (or secrecy). Related to disclosure of information.
2. Integrity, related to modification of information.
3. Availability, related to denial of access to information.

For example, confidentiality is concerned with preventing an employee from finding out the boss's salary; integrity, with preventing an employee from changing his or her own salary; and availability, with ensuring that paychecks are printed on time [14].

Bell and LaPadula developed lattice-based access control models to deal with information flow in computer systems.

Information flow is clearly central to confidentiality and also applies to integrity to some extent. But its relationship to availability is tenuous at best. Hence, these models are primarily concerned with confidentiality and can deal with some aspects of integrity.

2.4. Lattice Based Access Control

Lattice-based access control models were developed in the early 1970s to deal with the confidentiality of military information. In the late 1970s and early 1980s, researchers applied these models to certain integrity concerns [14]. Later, application of the models to the Chinese wall policy, a confidentiality policy unique to the commercial sector, was demonstrated. A balanced perspective on lattice-based access control models is provided. Information flow policies, the military lattice, access control models, the Bell-LaPadula model, the Biba model and duality, and the Chinese wall lattice are reviewed.

2.4.1. Orders and Lattice

A lattice can be mathematically defined as a:

1. partial order of a set: binary relation that is,
   a. transitive: $a \geq b \; and \; b \geq c \; then \; a \geq c$
   b. reflexive: $a \geq a$
   c. anti-symmetric/acyclic: $a \geq b \; and \; b \geq a \; then \; a = b$
2. Total order: like a chain (either $a \geq b \; or \; b \geq a$).
3. Lattice: every subset has a least upper bound (LUB), and a greatest lower bound (GLB).

2.5. A Security Lattice for MLS

A Security Lattice can represent Multi Level Security represented in the form of MLS labels as shown in Figure 2.5.

1. The security lattice is a graphical representation of the dominance relationship between all labels in the system.
2. For this example, the system has four sensitivity/clearance labels in this order of sensitivity $TS > S > C > U$ and 2 compartments (A and B).
3. If a path exists from one node to a second node then the label associated with the first node strictly dominates the label associated with the second node.
4. Information is permitted to flow from the first node to the second node.
5. Labels towards the top of the diagram have a higher sensitivity/clearance.
6. Labels towards the right side of the diagram have more categories (need to know).
7. The special label $TS{:}\,AB$ is referred to as System High because it dominates all other labels in the system and information may flow to it from any label in the system.
8. The label U is referred to as System Low because it is dominated by all labels in the system and information may flow from it to any other label on the system.

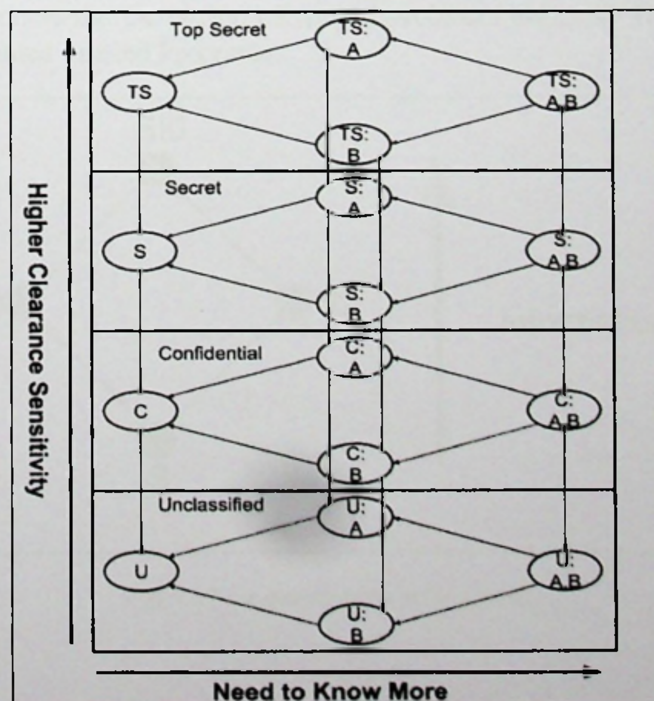


Figure 2.6 MLS Security Lattice

2.5.1. Information flow in a lattice with Multi Level Security

As shown in Figure 2.6 below, there are 4 running processes in a system:

1. The lowest one (1st process) is labelled as '***s***'.
2. It is at the bottom of the lattice, so it can be read, but not written by processes in other parts of the lattice.
3. The 2nd process is assigned the label ***SO***.
4. The 3rd process is assigned the label ***SI***.
5. The 4th process is labelled as '***SIO***' which is used to store data that cannot be read by any other processes in other parts of the lattice (although they can write it).
6. Since there is no relationship between SI and SO, no information can flow between them.

This is a problem, because if we need the 2nd process to communicate with 3rd process in both ways, then we need to get data from 2nd process to 3rd process and back again in a controlled, secure fashion. The information flows permitted by the lattice are too restrictive to build a useful system.

To address this problem several privileges are introduced that a process can possess to allow it to override the information flows permitted by the lattice security model. To relay information across the lattice, privileged processes are used. These privileged processes are called Trusted Processes.

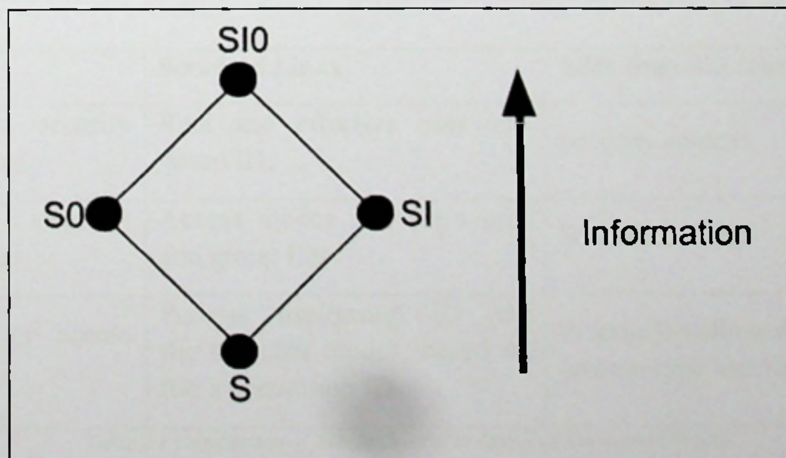


Figure 2.7 The security lattice in Virtualvault

2.6. SELinux Overview

2.6.1. Introduction

SELinux policy has several MAC mechanisms such as Type Enforcement which can together couple with MLS Policy. SELinux is based on the new security architecture Flask, based on flexible type of enforcement mechanism [15].

The primary advantage of this mechanism in comparison to traditional Linux security is the advantages over the DAC model. Let's consider a Linux process and a file where each process has a real and effective user and group IDs. Access to these attributes is controlled by the kernel and set by login process and setuid program [16]. Files are associated with inodes, containing information about access mode bits and file user and group IDs. This information includes three bits, defining read, write and execute operations for file owner, group and all others. These bits are then used to decide what users and groups may or may not access this file. Access controls enforced by SELinux present conceptual difference. Access control attributes are determined by special construction, called security contexts. For example, when a process attempts to access the file, traditional Linux DAC controls user/group of the process, the file's access mode and user/group IDs and, finally, makes decision, based on these data. Access control in SELinux doesn't exclude all these checks, but only continues this procedure by checking security contexts that can be defined for both the process and the file. So therefore, SELinux access control decisions are based upon types which is called Type Enforcement. The following Table 2.1 is a summary of access controls in traditional Linux and SELinux.

	Standard Linux	**Adds from SELinux**
Process security attributes	Real and effective user and group IDs	Security context
Object security attributes	Access modes and file's user and group IDs	Security context
Basis for access control	Process user/group ID and file's access modes, based on file's user/group ID	Permission allowed between process type and file type

Table 2.1 Comparison of Access Control in Standard Linux and SELinux

Therefore, we can make a conclusion that SELinux only presents additional means to build flexible access controls, but doesn't cancel standard controls, enforced by DAC. For example, if SELinux allows a process to read a file, but in DAC this access mode is denied, then a process would not be able to read that file and vice versa. Using

SELinux, we have ability to define various objects, permissions, and rules that result in one complex system, called security policy.

2.6.2. Subjects and Objects

SELinux has two distinct two frequently used words: Subjects and Objects. Subject is an entity, initializing an action. As an action, we consider some kind of performed operation. Typically, subjects perform some operations on objects, or even on other subjects. Subjects are the actors in computer systems. Initial think is that users would be the subjects, however, processes are the true actors [17].

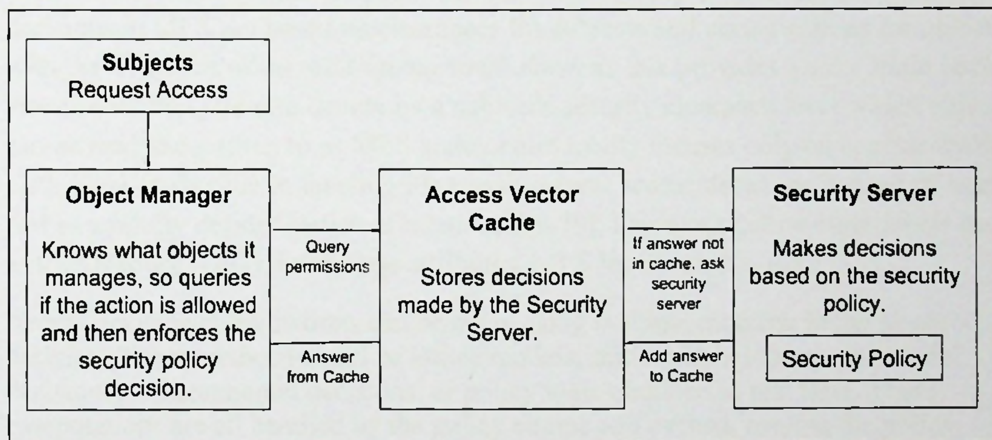


Figure 2.8 High Level SELinux Components

Objects are system resources such as files, sockets, links, database objects, devices etc. Similar objects are grouped to an object class. In some cases, processes could appear as objects when other processes perform some operations on them. Main SElinux components as shown in Figure 2.8 above can be defined as follows [18]:

1. An action is performed on an object and is initiated by a subject (e.g. a process reads a file).
2. An Object Manager is aware of the required actions (e.g. read) and the objects involved (E.g. file) and is able to enforce those actions.
3. A Security Server makes decisions, based on subject rights and security policy rules.
4. A Security Policy is enforced with the rules, using the SELinux policy language.
5. An Access Vector Cache (AVC) improves system performance by caching security server decisions.

LIBRARY
UNIVERSITY OF MORATUWA, SRI LANKA
MORATUWA

In short, each action in SELinux is identified by subject, object and the performed operation. Security server checks whether a given request/response is in access vector cache, and if not, then consults access rules, defined in security policy logic [15]. Finally, it makes decision and stores a new entry to AVC.

2.6.3. Flask Architecture

Flask was developed to overcome problems with the MAC architecture as traditional MAC is closely integrated with the *multi-level security* (MLS) model. Access decisions in MLS are based on clearances for subjects and classifications for objects, with the objective of *no read-up, no write-down* as this provides a very static lattice that allows the system to decide by a subject's security clearance level which objects can be read and written to as MLS architecture totally focuses only on confidentiality [19]. Flask is flexible in labeling for transition and access decisions instead of being tied to a rigidly defined lattice of relationships [9], Flask can define other labels such as user identity (UID), roles, type attributes, MLS levels, and so forth [19].

Access decision computations can be made using multiple methods in the same decision. These methods could be lattice models, static matrix lookups, historical decisions, environmental decisions, or policy logic obtained in real time. These computations are all handled by the policy engine and cached, leaving the policy enforcement code available to handle requests [19].

The flexibility of flask architecture as shown in Figure 2.9 provides where any of these subsystems can be swapped out for a new or different system without the other systems being aware of the changes done on the other components [19].

TH 4243

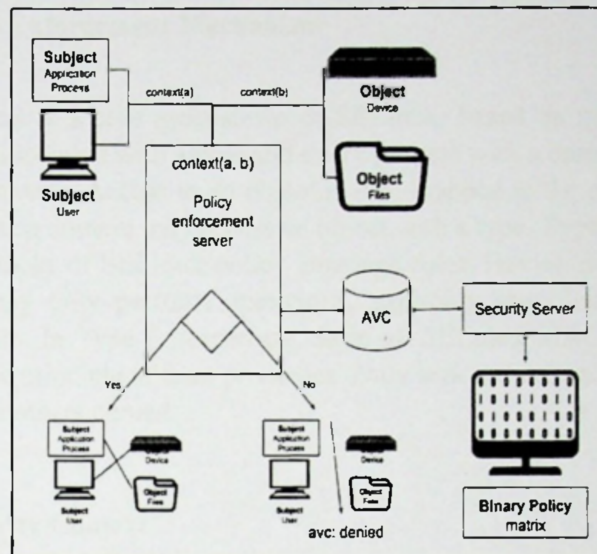


Figure 2.9 Flask Architecture

In Figure 2.8 based on operation occuring in the flask architecture, standard DAC permission already has passed [19]. The operation can be reading from or writing to a file/device, transitioning a process from one type to another type, opening a socket for an operation, delivering a signal call, and so forth.

1. A subject, which is a process, attempts to perform an operation on an object, such as a file, device, process, or socket.
2. The policy enforcement server gathers the security context from the subject and object and sends the pair of labels to the security server, which is responsible for policy decision making.
3. The policy server first checks the AVC and returns a decision to the enforcement server.
4. If the AVC does not have a policy decision cached, it turns to the security server, which uses the binary policy that is loaded into the kernel during initialization. The AVC caches the decision, and returns the decision to the enforcement server, that is, the kernel.
5. If the policy permits the subject to perform the desired operation on the object, the operation is allowed to proceed.
6. If the policy does not permit the subject to perform the desired operation, the action is denied and written to /var/log/audit/audit.log file [19].
7. With the security server handling the policy decision making, the enforcement server handles the rest of the tasks. In this role, you can think of the enforcement code as being an *object manager*. Object management includes labeling objects with a security context, managing object labels in memory, and managing client and server labeling [9].

2.6.4. Type Enforcement Mechanism

Type Enforcement is a core mechanism of SELinux, based on types and domains. Every object is associated with a type and every process with a corresponding domain (i.e. type of a process). Access to an object is then mapped to the problem whether a subject with a given domain may access an object with a type. Types and domains are basic building blocks of SELinux policy language rules. Having defined these rules, every subject may only perform operations, explicitly specified in the scope of associated domain. In Type Enforcement, as in all SELinux MAC mechanisms, the key concept is the principle of least privileges. Only actions, allowed by the policy can be performed, all others denied.

2.6.5. Security Context

SELinux uses security contexts that can be assigned for all subjects and objects in a system. It is defined as SELinux user, role, type and an optional security level as:

```
user:role:type[:level]
```

Roles are used to have access to one or more types. User represents SELinux user identity and can be associated to one or more roles. An optional field is a level and can be only present if the policy supports MLS or Multi-Category Security (MCS).

2.6.6. Types and Attributes

SELinux already has defined types but we may declare own types and assign them some set of permissions. An attribute is an efficient way to group types with similar features [16]. Every type may have one or more attributes and each attribute can be associated with one or more types. By assigning some attribute to the type, we grant it with all privileges that an attribute has. Before using an attribute, it should be declared first. The following rules declare the attribute file_type and assign it to the type etc_t taken from the SELinux reference policy [20].

```
[root@ishara-msc-research refpolicy]# cat
../refpolicy/policy/modules/kernel/files.te
# Attribute declaration:
attribute httpd_server_domains;
# Type declaration:
type httpd_t;
# Association with typeattribute statement:
typeattribute httpd_t httpd_server_domain;
```

2.6.7. Access Vector Rules

The AV rules determine what is allowed for processes to perform. The common syntax of these rules is the following [16]:

```
rule_name source_type target_type : class perm_set;
```

In the following example process with the domain ***httpd_t*** is allowed to read and create files with the type ***httpd_sys_content_t***:

```
#Taken from SELinux refpolicy
[root@ishara-msc-research refpolicy]# cat
../refpolicy/policy/modules/contrib/apache.te
allow httpd_t httpd_sys_content_t: file { read create };
```

2.6.8. Domain Transition

Every process inherits the domain of its parent process which implies if a process running in *staff_t* domain spawns a child process it will operate in the same domain [1]. For example, as shown in Figure 2.10, it would not secure if some untrusted program has access to the file */etc/shadow*. Therefore, trusted program should be executed in the corresponding domain (e.g. ***passwd_t***) and the only ability to access shadow file would result in getting access to this domain.

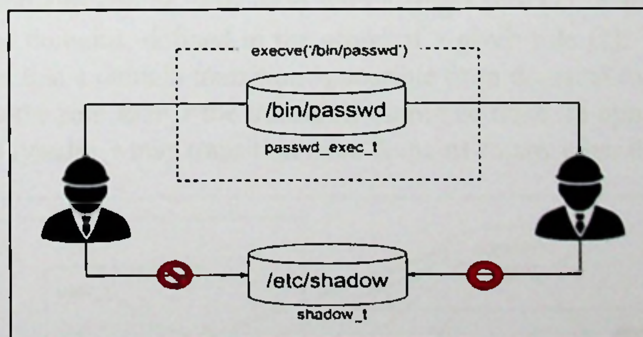


Figure 2.10 Domain Transition

The following rule allows process with staff_t domain to run in passwd_t domain after executing passwd program, having passwd_exec_t domain:

```
type_transition staff_t passwd_exec_t : process passwd_t ;
```

After loading this rule, the process will no be able to access file /etc/shadow directly, however domain transition makes it possible (see Figure 2.2).

2.6.9. Constraints

The purpose of Constraints is to provide additional restrictions for AV rules. Let us assume, the following rule:

```
allow staff_t passwd_t: process transition;
```

This simple rule gives a permission to the domain staff_t to transition to the domain passwd_t without checking any conditions. But below statement shows transition is only allowed if the source (i.e. process) role (r1) is equal to the target (i.e object) role (r2). If this condition is not satisfied, then transition is denied for all processes [33].

```
constrain process transition ( r1 == r2 )
```

Role Based Access Control

A number of domains can be associated with a single role [1]. A process may only transition to the domains, defined in the scope of a given role [1]. The Figure 2.11 shown indicates that a domain transition is possible from domain4 to domain5, but if the process has the role ***user_r*** the transition cannot be done. In opposite, processes, having the role sysadm_r may transition from domain1 to any other domain.

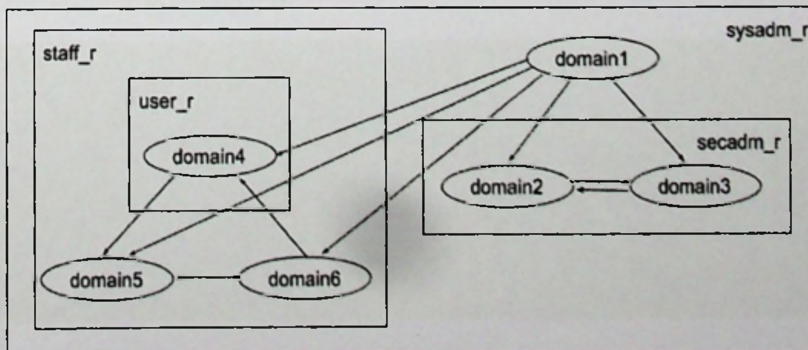


Figure 2.11 Conceptual Diagram of RBAC

Role Statements

Roles can be associated with corresponding types. Such association allows roles and types to coexist in the same security context [16]:

```
role user_r types { user_t staff_t };
```

This rule associates role user_r with two types; user_t and staff_t.

Below shows an allow rule for role transition [16]:

```
allow sysadm_r secadm_r;
```

The above rule allows a process with the role sysadm_r to "switch" to the role secadm_r. This rule specifies a default role change after executing some program. Let us consider the following rule:

```
role_transition sysadm_r passwd_exec_t secadm_r;
```

It changes the role ***sysadm_r*** of the process to the role **secadm_r** after executing a file with the type passwd_exec_t. This kind of rules are also often used to change the role of system daemons, where interaction with user is not required.

Users and Roles

SELinux users and Linux users are considered as two different identities. The idea of such approach was to make SELinux user, independent from its Linux representation [16]. Therefore, changes for Linux user do not affect SELinux user

For example, to check all available associations between SELinux users and roles one can use the following command:

Figure 2. 2.1 List of SELinux Users

In MLS policy we may also specify MLS level or range. The previous rule then can have transformed to the following:

```
user admin_u roles { sysadm_r secadm_r } level s0 range s0-s15:c0.c1023;
```

The same can be done using semanage:

```
# semanage user -a -R sysadm_r secadm_r -r s0-s15:c0.c1023 -s admin_u
```

Mapping between Linux and SELinux Users

A Linux user can be mapped to exactly one SELinux user. The easiest way to do it is to use semanage. The following command defines a mapping between SELinux user 'admin_u' and Linux user 'ishara':

```
# semanage login -a -s admin_u ishara
```

Audit Logs

Writing and configuration of SELinux policy would be quite difficult without logs. They help system administrator to fix different problems and analyze access denials. Two main types of audit events can be classified as follows [18]:

1. *SELinux-aware application events* - System errors, change of boolean states, setting of enforcing/permissive mode, relabeling [9] etc. All these events are logged by the SELinux kernel services and SELinuxaware applications.
2. *AVC (Access Vector Cache) audit events* - Access denials, generated by AVC system.

Both types of event messages can be stored in two places: */var/log/messages* and */var/log/audit/audit.log.*

```
# ausearch -m avc -ts 13:20
type=AVC msg=audit(1343992881.544:1053): avc: denied {read}
for pid=1640 comm="httpd" name="index.html" dev=dm-1
ino=264900 scontext=system_u:system_r:httpd_t:s0
tcontext=system_u:object_r:httpd_sys_content:s1 tclass=file
```

The above appeared as a violation of MLS policy when process httpd, having security context *system_u:system_r:httpd_t:s0* was attempting to read a file index.html labeled as *system_u:object_r:httpd_sys_content_t:s1*.

Using program *audit2allow*, it is possible to transform AVC messages to allow rule, as follows:

```
# ausearch -m avc -ts 13:20 | audit2allow
#============= httpd_t ============
#!!!! This avc is a constraint violation. You will need to add an
attribute to
either the source or target type to make it work.
#Constrain rule:
allow httpd_t httpd_sys_content_t:file read;
```

Adding generated rule to the policy, sometimes, solves the problem. But in the case of global costraint violation this would not help. As it is shown on our example, the actual reason of access denial was the violation of MLS constraint ("no-read-up" property). In such cases we can use special attributes, that should be assigned to either the process or object type.

SELinux Booleans

SELinux can be configured without knowledge of policy writing without reloading the policy using SELinux booleans feature which takes the values on or off as shown below:

```
# semanage boolean -l
SELinux boolean Description
ftp_home_dir ->off  Allow ftp to read and write files in the user
home directories ...
```

The command above lists booleans, their current value and shows a brief description of each boolean. The purpose of booleans is to simplify work of users and system administrators in policy configuration.

Object Labelling

For SELinux policies to correctly work, all its objects must be associated with security contexts. Every policy installation requires labeling/relabeling of the file system to assign initial security contexts to objects [9]. After an installation, security contexts can be changed.

It should be noted that there is a difference in the time of validity of the label, assigned by the programs. For example, *chcon* program provides only temporary changes (rebooting will revert back the labels) and semanage fcontext saves and writes into the SELinux policy server. For the most object classes SELinux implements the concept that created object inherits security context of the creating process, containing object or combination [16]. For example, newly created file obtains a type from the directory

containing it, hard-coded role (*object_r*) and the SELinux user of the creating process. Processes obtain their contexts in two basic ways: inherit from parent process or change it using domain/role transition [18].

2.6.10. SELinux Policies

SELinux is configured via policies. Policy is a set of rules that control an access to objects of the system. All policies can be classified using the following classification [21]:

1. Based on the source code: Example, Reference or Custom policy
2. Further descriptions of the source code can also lead to sub-classification: Monolithic, Base Module or Loadable Module
3. Based on functionality: targeted, mls, strict or minimum
4. Based on language statements: Modular, Optional or Conditional

Reference policy is currently the standard SELinux policy source and using the source of Reference policy, we may install policies with different security goals such as **minimum** policy, **targeted** policy, **Strict** policy and **MLS** policy [22].

MLS policy is further development of strict policy, but it enforces Bell-LaPudula model, using security levels [22].

Every policy works in two modes: permissive and enforcing (or can be fully disabled). In permissive mode every operation is allowed. Denied operations are immediately notified in logs. This mode is often used in the policy development process (so called "debugging" mode). Enforcing mode enables SELinux policy, denying access and logging actions.

For studing MLS policy, we may install it together with the Reference policy and do a relabel of the file system as follows.

```
# yum install selinux-policy-mls
```

After an installation has been completed, it is necessary to turn the policy on in the configuration file /etc/selinux/config, using an option SELINUXTYPE=mls. After that, relabeling of the system must be done [9]. The following command will relabel the system with new labels after reboot:

```
# touch /.autorelabel && init 6
```

2.6.11. Policy Modules

Policy modules are currently basic components of the Reference policy. Most of type enforcement rules are implemented within modules. They control behavior of different applications, services, devices and many other system elements [23].

To create a module, we must define three files: .te, .fc and. if. The common structure of the .te file may look as the following [23].

```
policy_module(<module_name>, <module_version>)
gen_require(`
<required types, attributes, object classes ...>
') ...
<declarations>
...
<policy rules>
...
```

Using gen_require macro we can specify the necessary selinux components, defined in other modules. In the file contexts file (.fc) we can specify objects and the default labels they should have. For example, the following line defines the security context system_u:object_r:myapp_exec_t:s0 for the binary file myapp:

```
[root@ishara-msc-research doc]# cat
/root/refpolicy.wiki/files/refpolicy/doc/example.fc
/usr/sbin/myapp        -
-    gen_context(system_u:object_r:myapp_exec_t,s0)
```

The interface file *(.if)* defines macros that can be used by other modules. For this purpose, SELinux uses m4 macro language. It simplifies future reusage of the defined rules. Let us assume the policy module for an application, running in myapp_t domain, having entrypoint domain *myapp_exec_t*. The following demonstrates how an interface that allows other domains to do a domain transition to *myapp_t*, by executing a program labeled as *myapp_exec_t* [23].

```
#Taken from SELinux reference policy []
[root@ishara-msc-research doc]#  cat
refpolicy.wiki/files/refpolicy/doc/example.if
interface(`myapp_domtrans',`
	gen_require(`
		type myapp_t, myapp_exec_t;
	')
	domtrans_pattern($1,myapp_exec_t,myapp_t)
')
```

This interface defines new macro *myapp_domtrans* where $1 represents the parameter passed to this macro. It calls *domtrans_pattern* macro that takes $1 as the first

parameter and allows it to transition to the specified domain [23] (in our case myapp_t with the entrypoint myapp_exec_t).

Finally compile and make the module while it checks the syntaxes as well [23].

```
# make -f /usr/share/selinux/devel/Makefile
```

To load the module into the kernel:

```
# semodule -i <module_name>.pp
```

To ensure that the policy module is in the kernel we may check the list of loaded modules.

```
# semodule -l | grep -i <module-name>
```

2.6.12. Summary of SELinux Overview

In this chapter we studied how SELinux implements access control by explicit specifying of rules which creates policies. Most of each policy rules are created from type enforcement rules. Every object is associated with a security context, represented with user, role, type and optionally security level. To work with MLS, we may install base MLS policy, but for deeper analysis we need the Reference policy. Every policy works in permissive and enforcing mode or can be completely disabled. For debugging purposes permissive mode is more suitable, while enforcing enables denying access and can be viewed through selinux audit logs. Custom policies can be loaded into the security server using existing macros by creating & loading SELinux modules into the security server.

2.7. SELinux Multi-Level Security

Multilevel Security in SELinux represents a new form of Mandatory Access Control. It is built upon Type Enforcement but extends it with new features. The Bell–LaPadula model (BLP) was chosen as the base. This model focuses on data confidentiality [24], in contrast to Biba model which is oriented on the integrity protection [25]. In terms of the BLP model all processes have their security level, allowing them to access objects with the same security level (read and write). Moreover, as shown in Figure 2.13 a process may read objects with lower security level and write to objects with higher security level, but not the reverse [26]. These concepts are also known as "no-read-up" and "no-write-down" rules.

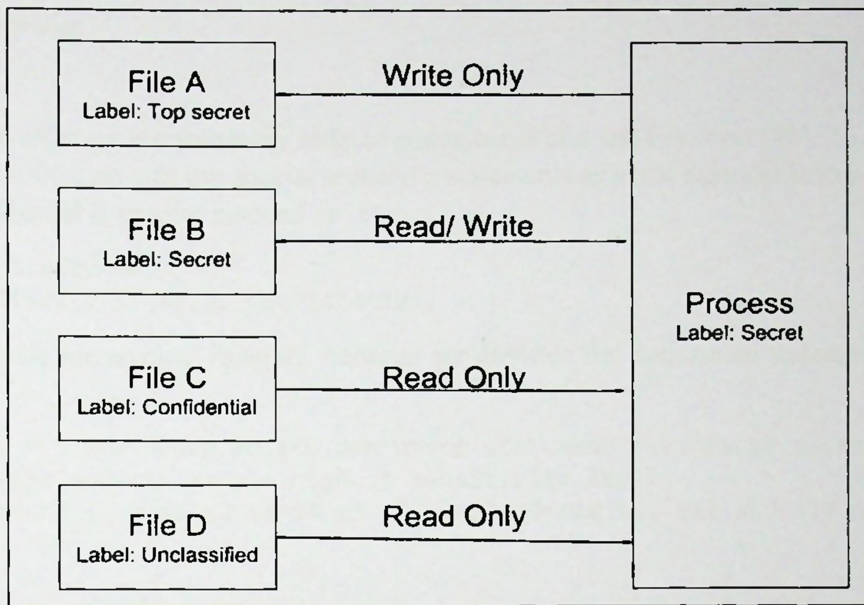


Figure 2.12 Bell Lapadula Model

2.7.1. Security Levels

Security level consists of 16 sensitivities from $S0$ to $S15$ and 1024 categories from $C0$ to 1023. Sensitivities can be also understood as classifications (e.g. Confidential, Secret, Top Secret, etc.) and categories as compartments (e.g. Finance, Marketing, Personnel, etc.

In the MLS system, extended security context is represented in the following format:

```
user:role:type:LowSensitivity[:Categories]-
HighSensitivity[:Categories]
```

Security level consists of sensitivity and zero or more categories, but it can be also represented as a range. Low security level consists of low sensitivity and, optionaly, a set of categories. It is applied for subjects and objects and is also known as the Current or Effective security level. The lowest security level in the system is defined as SystemLow and doesn't contain any categories (that is s0). Category set can be defined in two ways. Using comma, we may list each category as: $c0, c1, c2, c3$. The same could be done, using inclusive operator "dot": c0.c3. High security level consists of high sensitivity and optional set of categories. It is also known as the Clearance. The highest level SystemHigh consists of the highest sensitivity (e.g. s15) and all available categories (e.g.$c0. c1023$). Security levels are defined using special level statements, as follows:

```
level s2:c0.c5
```

Sensitivities

Sensitivities are hierarchically ordered components of a security level [16]. To define sensitivities, we may use special sensitivity statements as in the example below where Confidential is an alias mapped on '$s1$'.

```
sensitivity s0;
sensitivity s1 alias Confidential;
```

To create hierarchical relations between sensitivities the dominance statements are used.

```
# The MLS Reference Policy dominance statement defines s0 as the
# lowest and s15 as the highest sensitivity level:
dominance { s0 s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11 s12 s13 s14 s15 }
```

Categories

In contrast to sensitivities, we cannot compare categories as they are not hierarchically related [16]. Their values are defined, using the category statements as follows:

```
category c0;
category c1 alias Finance;
```

Categories can be used not only in the MLS policy. For example, in the targeted policy security level is represented with one sensitivity $s0$ and zero or more categories which is refered to as Multi-Category Security (MCS) which can be used to further constrain DAC and TE SELinux policies.

Let us look at simple examples of access control, based on MCS. At first, let us define a user with the permission to access three categories $c0, c1$ *and* $c2$:

```
[root@localhost]# chcat -l - +c0,+c1,+c2 admin
```

The command above defines this permission for Linux user admin. To check the categories accessible for this user the following command can be used:

```
[root@localhost]# chcat -L -l admin
admin: c0,c1,c2
```

Then, let us create an arbitrary file and label it with the same set of categories:

```
[root@localhost]# echo 'File labeled with the categories' >
/tmp/file
[root@localhost]# chcat -- +c0,+c1,+c2 file
admin's shell:
[admin@localhost]$ cat /tmp/file
File labeled with the categories
```

Now user admin may successfully read the file. But removing at least one category from the user's security level or adding at least one new category to the file's level will deny an access. Let us add a new category to the file and test the results:

```
[root@localhost ]# chcat -- +c3 /tmp/file
admin's shell:
[admin@localhost tmp]$ cat /tmp/file
cat: /tmp/file: Permission Denied
```

As we may see, at this moment permission is denied. It could be explained by the fact, that a set of categories of an object (in our case it is a file) must be completely subsumed by a set of categories that a user (user's process) may access [1]. In our case it doesn't hold as the user admin cannot access the category *c*3.

2.7.2. Security Level Translation

Sensitivities and categories can be given human readable names. The first way is using aliases in sensitivity and category statements. The other way is using mcstrans service. The configuration file used by this service is typically located in /etc/selinux/mls/setrans.conf.

To set new translations it is enough to modify this file, as follows:

```
[root@ishara-msc-research refpolicy]# echo "s3=TopSecret" >>
/etc/selinux/mls/setrans.conf
[root@ishara-msc-research refpolicy]# service mcstrans restart
[root@ishara-msc-research refpolicy]# chcat  -L | grep -i top
s3                                TopSecret
```

2.7.3. mlsconstrain Statements

The basic logic of the MLS policy is defined by special mlsconstrain statements. They allow to restrict permissions for object classes, based on security levels of a source process and a target object. Security level is represented by its low ($l1$ for the source and $l2$ for the target) and high value ($h1$ for the source and $h2$ for the target). Formal syntax of these statements is the following [18]:

```
mlsconstrain class_set perm_set expression;
```

Parameter class_set represents one or more object classes (e.g. {file dir lnk_file}). perm_set defines permission set (e.g. {read gettattr execute}). The main part of the statements is boolean expression, that can be determined as follows [18] as shown in Table 2.2.

```
( expression : expression )
| not expression
| expression and expression
| expression or expression
| u1 op u2
| r1 role_op r2
| t1 op t2
| u1 op names
| u2 op names
| r1 op names
| r2 op names
| t1 op names
| t2 op names
| u3 op names
| r3 op names
| t3 op names
```

UNIVERSITY OF MORATUWA LIBRARY

Where:

Symbol	Symbolic representation
$t1, r1, u1, l1, h1$	Source type, role, user, low level, high level
$t2, r2, u2, l2, h2$	Target type, role, user, low level, high level
op	== | ! =
oprl	== | ! = | *eq* | *dom* | *domby* | *incomp*
==	Set member of or equivalent
! =	Set not member of or not equivalent
eq	equivalent
dom	dominates
domby	dominated by
incomp	incomparable
names	*name* | { *name_list* }
name_list	{ *name_list* } | *name*

Table 2.2 MLS Constraint Table

In general, formal syntax is like SELinux constrain statements [27]. Operators, defined by oprl can be also used to compare roles. The following mlsconstrain statement implements a simple "**no-read-up**" rule:

```
mlsconstrain file {read getattr execute}
( (l1 dom l2) or
(t1 == mlsfilereadup) );
```

It is applied for the file object class. Permission set is represented by read (file reading), getattr(getting file's attributes) and execute(file execution). The first part of the boolean expression states that low level of the source process must dominate low level of the target object. In other words, processes may read files only on its Current or lower security levels. In the second part of expression we defined an exception from the rule (that is processes, having access to the attribute mlsfilereadup may violate "no-read-up" rule, whatever security level they have). In similar way, we may define "no-write-down" rules, but instead of dom operator we should use domby (that is the source low level must be dominated by the target low level) and list the corresponding permissions and objects.

2.7.4. mlsvalidatetrans statements

This type of statements is only applied for file-related object classes to control the ability to upgrade/downgrade security level. The formal syntax is like mlsconstrain statements, but there is a bit difference [27] as shown in Table 2.3.

```
mlsvalidatetrans class_set expression;
```

Symbol	Symbolic Representation
class_set	One or more file type object classes
expression	A boolean expression of the constraint
u1,r1,t1,l1,h1	Old user, role, type, low level, high level
u2,r2,t2,l2,h2	New user, role, type, low level, high level
u3,r3,t3,l3,h3	Process user, role, type, low level, high level

Table 2.3 MLSConstraint Conditional Operators

Let us look at the following example to understand how these statements work:

```
mlsvalidatetrans file
(( l1 eq l2 ) or
(( t3 == mlsfileupgrade ) and ( l1 domby l2 )))
```

The first part of the boolean expression (l1 eq l2) states that file's security level can be changed if its current(old) low level is equal to the new file's low security level. The second part or ((3 == mlsfileupgrade) and (l1 domby l2)) claims: or the process type must be privileged with the mlsfileupgrade attribute and file's current low security level must be dominated by new file's low security level. It was an example of file's upgrading. In similar way we can also define an expression for file downgrading. In this case it is enough to change only the second part of the expression: or ((3 == mlsfiledowngrade) and (l1 dom l2))

2.7.5. Privilege Management

MLS policy provides a flexible mechanism of privilege management, based on special type attributes. These attributes can be used to obtain access privileges for different objects. For example, the attribute mlsfileread can read files at all levels. To obtain privileges of this attribute it is enough to define the following rule:

```
typeattribute mytype_t mlsfileread;
```

The same could be done by using the macro:

```
mls_file_read_all_levels(mytype_t)
```

All attribute names and interface calls start with "mls" [20]. Table 2.4 presents some important MLS attributes with their descriptions:

Attribute	Description
mlsfileread	Grant MLS read access to files not dominated by the process Effective SL
mlsfilereadtoclr	Grant MLS read access to files dominated by the process Clearance SL
mlsfilewrite	Grant MLS write access to files not equal to the Effective SL
mlsnetread	Grant MLS read access to packets not dominated by the process Effective SL
mlsipcwrite	Grant MLS write access to IPC objects not equal to the process Effective SL
mlsprocread	Grant MLS read access to processes not dominated by the process Effective SL
mlstrustedobject	Grant MLS read/write access to objects which internally arbitrate MLS

Table 2.4 SELinux Attributes

The mechanism of using attibutes is an ellegant approach because it allows us to implement different MLS information flows without global policy overrides [28]. All limitations, defined by mlsconstrain and mlsvalidatetrans statements specify these attributes and the way, they can be used, to bypass those limits.

UNIVERSITY OF MORATUWA LIBRARY

2.7.6. Complexitiy of SELinux policies

SELinux policies are developed and maintained by security administrators, they often become quite complex, and it is important to carefully analyze them in order to have high assurance of their correctness. There are many existing analysis tools for modeling and analyzing SELinux policies with the goal of answering specific safety and functionality questions. The policies typically are comprised of thousands of policy statements; this makes policy development and analysis very difficult. Even when a policy is considered both safe and functional, each addition, deletion or modification of the policy has the potential to break the baseline the complexity of the SELinux policy language makes analyzing SELinux policies and even implementing policies very difficult

Information flow is about the reachability of a resource from another resource where some information is transferred by performing an operation. For example, there is an information flow between a subject $S1$ to a subject $S2$ if $S1$ can perform a write operation on some objects on which $S2$ can perform a read operation.

2.7.7. SELinux policy analysis tools

There are number of tools written in different higher-level languages intending to analyze SELinux policies of various SELinux security models. These tools have various capabilities to perform information flow analysis based on type enforcement, some tools have the capability to deduce whether a TCB is integrity protected. Some common tools are APOL [29], GOKYO [30] and SEEDIT [31].

Refer to Appendix A for a comprehensive list of comparison of SELinux policy analysis tools.

2.7.8. Problems with SELinux analysis tools

SELinux policy language itself complicates both the implementation of policies as well as the ability to analyze them. As a result, many tools are complex, and it is difficult to establish the correctness of the analyses they perform. One problem with these tools is that they do not use the same criteria in support of each other; moreover, as mentioned, analysis tools try to provide some other intermediate language for SELinux security administrators. Although these extra facilities can help with writing various queries, they require equally complex semantics. Furthermore, existing SELinux analysis tools barely scratch the surface and only offer the possibility of doing simple queries.

2.7.9. SELinux Challenges and Proposed Solution

The SELinux policy language doesn't have formal semantics. Its semantics is given in terms of a natural language description. Expressing the semantics of an access control policy language in a natural language (e.g. English) results in ambiguity in the specification of behavior of policy statements. Consider the last three lines of Figure 2.14 which protects the entrypoint access [24] of the app_t domain. Removing any one of these rules will break the intended protection because for a domain transition to occur, all three rules are required. The first rule provides execution permission for the domain sysadm_t on the file with the type app exec t; the second rule provides an entrypoint for the domain app_t; the third rule provides a type transition to the new type app t from the current type sysadm_t. The fact that SELinux rules are so fine-grained adds to the complexity of SELinux. Both writing and analyzing policies are difficult tasks. It is hard for administrators to express the desired protection using such a low-level language.

```
require {
attribute domain;
attribute file_type;
attribute exec_type;
type sysadm_t;
attribute sysadm_r;
class process transition;
role sysadm_r;
}

type app_t;
typeattribute app_t domain;
type app_exec_t;
typeattribute app_exec_t file_type;
typeattribute app_exec_t exec_type;

role sysadm_r types app_t;     }
type_transition sysadm_t
app_exec_t : process app_t;
allow sysadm_t app_exec_t : file (getatr execute);
allow app_t app_exec_t : file entrypoint;
allow  sysadm_t app_t : process transition;
```

Adding types and attributes that are required by the rules

Declaring new types and classify them by attribute

Assigning roles to types

Defining default transition and its required access

Figure 2.13 App Program Security Policy Rules in SELinux

3. METHODOLOGY

The research methodology will utilize the concepts of lattice-based access control as indicated in **Sections 2.4 and 2.5** on information flow policies for an MLS based environment. The security policies in SELinux MLS is based on a customized version of the Bell Lapadula model as explained in Section **2.6.10** which is already made available in the SELinux security server. i.e: SELinux security server enforces a customized version of the Bell-Lapadula model (No read up, no write down and writes can the same sensitivity level).

Two widely used common network services which runs on linux environments is to be chosen to run on an SELinux MLS environment. The security labels such as the types for the subjects are chosen upon based on the SELinux policy mlsconstraints statements to satisfy the no-readup and no-writedown (and write-equal property) on the corresponding subjects as explained in **Section 2.7.3**.

The security levels for the two services are determined in the form of an inequality using a python script. The python script deduces the security levels depending on the need of information flow in the security lattice for the subjects in order to satisfy the simple property and the customized *-property.

Once these security levels are determined, the security administrator has the option to assign these security levels for the subjects and the objects associated with the two services in the form of SELinux modules as shown in **Section 2.6.11** which could be loaded into the current SELinux policy server.

3.1. Lattice labels for SELinux

The SELinux security server has a set of sensitivities and categories loaded into the security server which can be applied over any subject and object in a SELinux MLS environment, the possible security labels which could be labelled on any subject or object can be deduced using the following methodology:

Let $SVC1$ and $SVC2$ be the two services which are intending to run on the SELinux MLS enabled environment

Let's assume that $SVC1$ and $SVC2$ are run with labels which are composed of both sensitivity levels and category levels which denotes the following:

$L1$ and $L2$ be the sensitivity levels and, $C1$ and $C2$ be the categories for Services $SVC1$ and $SVC2$ respectively which is composed of the following categories:

$$C1 = X, Y \text{ and } C2 = P, Q$$

From **Section 2.4.2,** the partial order on security classes is called dominates:

$(L1, C1) \geq (L2, C2)$

$iff\ L1 \geq L2$

$\& \{P, Q\} \subseteq \{X, Y\} \ ---> \ C2 \subseteq C1$

1. The set of Security levels available in a SELinux MLS environment is 16 which is finite as per the **Section 2.7.1**.
2. There exist a greatest lower bound and a least upper bound as the categories available in a SELinux as per Section **2.7.1** is 1024.

Therefore, as per **Section 2.4.3**, the lattice formed for the information flow between $SVC1$ and $SVC2$ in the SELinux MLS environment will form a finite lattice.

In a SELinux MLS environment as per **Section 2.7.1**, 'S' denotes for the sensitivity level and 'C' denotes for the category where:

$S = \{S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15\}$

$S0 < S1 < \ldots\ldots S15$

and

$C = \{C0, C1, C2, \ldots., C1023)$

Elements of 'C' is incomparable and are not hierarchically related

Therefore the $Least\ Upper\ Bound(LUB)$ and the $Greatest\ Lower\ Bound(GLB)$ for the finite lattice formed for information flow in an SELinux MLS enabled environment can be deduced as:

$\boldsymbol{LUB}\ ((S15, \{C0\}), (S0, \{C0.C1023\})) \ = \ (S15, \{C0.C1023\})$

$\boldsymbol{GLB}\ ((S15, \{C0\}), (S0, \{C0.C1023\})) \ = \ (S0, \{C0\})$

$where\ C0.C1023 = \{C0, C1, C2, \ldots., C1023\}$

From **Section 2.5** we can deduce the dominance relationship between LUB and GLB as:

$$S15.C1023 \ > \ S0.C1023$$

Where $S15.C1023$ is referred to as SystemHigh as it dominates all other labels in the system and information may flow to it from any label in the system. It is on the top most of the lattice and information flow is allowed from bottom to the top of the lattice.

$S0$ is referred to as SystemLow because it is dominated by all labels in the system and information may flow from it to any other label on the system. It is on the bottom of the lattice and information flow is allowed from bottom to the top of the lattice

Total number of label combination in a SELinux MLS system would be then:

$S0.C0, S0, C1, \ldots\ldots S0, C1023$

....

....

$S15.C0.C1023$

$= 16 * 2^{\wedge}1024$

3.2. Example of a dominance relationship

Using the partial order of SELinux labels using **Section 2.4.2**

i.e : $(L1, C1) \geq (L2, C2) iff\ L1 \geq L2\ and\ C2 \subseteq C1$

$(S15, C1.C5) > (S3, C1, C3)$ as the partial order formed on the security lattice formed on these SELinux labels, as;

$S15 > S3$ according to 'dominance' statement in the SELinux policy

$and\ C1, C3\ C\ C1.C5$

$where\ C1.C5 = \{C1, C2, C3, C4, C5\}$

3.3 Bell Lapadula Model review

As per the **Section 2.2.4,** the model focuses on confidentiality alone, following is an example of how information flows in the lattice with different sensitivities.

Assuming if we consider the following sensitivities labels.

$$S = \{S1, S2, S3, S4, S5\}$$

The simple property (No-Readup) implies that for a given set of subjects running in a system subjects on "sensitivity Level" $S3$ can NOT read objects on "Sensitivity level" $S4$ and $S5$.

The Star Property implies that subjects on "sensitivity level" $S3$ can NOT write to $S1$ and $S2$ and could write to the same level ($S3$).

However, the Star property also implies that the same subject $S3$ can write on $S4$ and $S5$, which means subjects with less sensitivity are allowed to write on subjects with higher sensitivity.

This is an integrity preservation issue where information flow may occur from untrusted information to trusted levels of information according to the Bell Lapadula model.

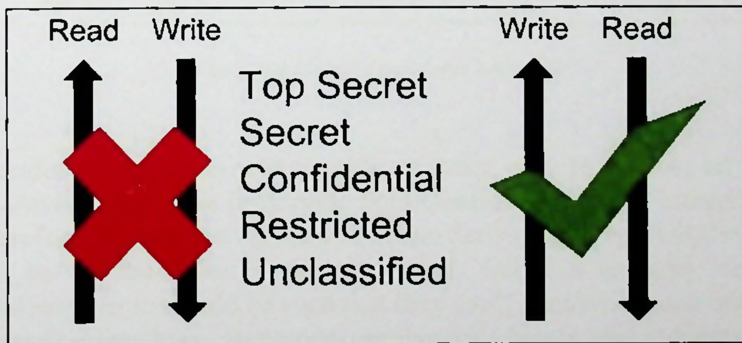


Figure 3.1 Bell-Lapadula Security Model

3.4 SELinux Policy alignment with Bell Lapadula model

Also, as we know using **Sections 2.6.10 and 2.7,** the SELinux security server implements and enforces a customized version of Bell-Lapadula model where it does the following.

1. No Read up.
2. No write down and no write up and allows only 'equal' writes which preserves integrity as well, this is demonstrated as follows.
3. No Write up, hence Integrity as well be preserved.

Figure 3.2 Modified Bell-Lapadula Model

4. Therefore, subjects would be able to write only to objects on the "SAME sensitivity level" thus preserving both Confidentiality and Integrity.
5. Therefore, the two services can be in two Sensitivity levels OR the two services can be in the same sensitivity level, but the category levels of the objects/subjects should be such that they can't read/write each other.
6. A service can have numerous number of objects and subjects loaded and running when a service up and running so it is not an easy task to figure out which objects need to access which object for the above two services because there are so many possible combinations.

UNIVERSITY OF MORATUWA LIBRARY

3.4.1 SELinux Policies facts in Customized Bell-Lapadula model

E.g.: If '$svc1$' is assigned on "Sensitivity Level": $S10$ and '$svc2$' is assigned with "Sensitivity Level" : $S11$, then according to the Bell-Lapadula model :

4. Then $S11$ has a higher sensitivity than $S10$
5. $S11$ can read $S10$ to $S11$ content
6. $S10$ can not read $S11$ content
7. $S10$ will write exactly the $S10$ content
8. $S11$ will write exactly the $S11$ content (not higher Sensitivities such as $S12$, S13, etc)

So as observed and as per the Bell-Lapadula model confidentiality & integrity between the two services will be preserved as it will not allow untrusted data to flow upwards in the lattice structure.

3.5 Overriding Bell Lapadula custom policy

The Bell lapadula security model is enforced in the SELinux policy language interms of '`mlsconstrain`' statements as per **Section 2.7.3**. There are various mlsconstrain statements loaded into the SELinux security server which are discussed in **Section 3.5.1 amd 3.5.2**.

3.5.1 Bell Lapadula Simple property in SELinux mlsconstrain statements

Following lines of mlsconstrain statements implements the Simple property, which is the no-readup rule in the Bell Lapadula model.

```
# the file "read" ops (note the check is dominance of the low
level)
 mlsconstrain { file } { read getattr execute }
        (( l1 dom l2 ) or
         (( t1 == mlsfilereadtoclr ) and ( h1 dom l2 )) or
         ( t1 == mlsfileread ) or
         ( t2 == mlstrustedobject ));
```

The above multi level security constrain states that:

File read/getattr/execute permission are only allowed if:

1. The process low-level ($l1$) dominates the file low-level ($l2$).

Or

2. The process type ($t1$) has the mlsfilereadtoclr (read-up-to-clearance) atttribute **and** the process high-level ($h1$) dominates the file low-level ($l2$).

Or

3. The process type ($t1$) has the mlsfileread (read-up) attribute

Or

4. The file type ($t2$) has the mlstrustedobject (e.g. `/dev/null`) attribute.

We can observe the boolean expression is coupled with '*and*' & '*or*' conditions.

Boolean Rule number 1 indicates the Bell lapadula simple property which is the no-readup property.

However since this is an 'or' condition, we can override and bypass the Bell lapadula simple property enforced by the SELinux security server and satisfy the conditions indicated by Boolean Rule number 2,3 or 4.

From **Section 2.6.6**, an attribute is nothing but a collection of similar types, for example the attribute 'mlsfileread' has the following 'types' mapped.

```
seinfo -amlsfileread -t | less
Types: 3158
   bluetooth_conf_t
   cmirrord_exec_t
   foghorn_exec_t
   jacorb_port_t
   sosreport_t
   etc_runtime_t
   ....
   ....
```

Therefore, using Boolean Rule number 3, we can imply that if the process has any of the above types of all the 3158 types which is mapped to attribute 'mlsfileread', then we can bypass the simple property (No readup rule) enforced by the SELinux security server.

It is in the hands of the Security Administrator to do label the process with this type so that (Hence we regard this as a trusted object) as this process labelled with '$t1$' will be able to read all files at all levels on the SELinux MLS enabled system.

According to **Section 2.7.5**, if the Security Administrator wants to bypass simple property and wants to allow process to read any file level, then it can be done as follows

```
typeattribute myreadtype_t mlsfileread;
```

Where '`myreadtype_t`' is the type of the process and hence once above is assigned, files of type '`mytype_t`' will be able to read files of all sensitivity levels (As per Section 3.1, $S0 \ldots .S15.C0.c1023$)

3.5.2 Bell Lapadula custom *-property in SELinux mlsconstrain statements

Similarly, as in **Section 3.5.1**, the following lines of `mlsconstrain` statements implements the custom *-property, which is the no-write down and write-equal rule (i.e: Hence a customized *-property) to overcome the integrity issues as discussed in **Section 3.3 and 3.4.**

```
# the "single level" file "write" ops
mlsconstrain { file } { write create setattr relabelfrom append
unlink link rename }
	(( l1 eq l2 ) or
	(( t1 == mlsfilewritetoclr ) and ( h1 dom l2 ) and ( l1
domby l2 )) or
	(( t2 == mlsfilewriteinrange ) and ( l1 dom l2 ) and ( h1
domby h2 )) or
	( t1 == mlsfilewrite ) or
	( t2 == mlstrustedobject ));
```

The above multi level security constrain states that:
File `read/getattr/execute` permission are only allowed if:

1. The process low-level ($l1$) is equal to the file low-level ($l2$).

   Or
2. The process type ($t1$) has the mlsfilewritetoclr (write-up-to-clearance) Attribute and the process high-level ($h1$) dominates the file low-level ($l2$) and the process low-level ($l1$) is dominated by the file low-level ($l2$).

   Or
3. The process type ($t2$) has the mlsfilewriteinrange attribute and process low-level (l1) dominates file low-level ($l2$) and process high-level ($h1$) is dominated by file high-level ($h2$).

   Or
4. The process type ($t1$) has the mlswrite attribute.

   Or
5. The file type ($t2$) has the mlstrustedobject (e.g. `/dev/null`) attribute.

We can observe the boolean expression is coupled with '*and*' and '*or*' conditions.

Boolean Rule number 1 indicates the Bell lapadula customized *-property which is the no-writedown and write equal property

However, since this is an 'or' condition, we can override and bypass the Bell lapadula custom *-property enforced by the SELinux security server and satisfy the conditions indicated by Boolean Rule number 2, 3, 4 or 5.

From **Section 2.6.6,** an attribute is nothing but a collection of similar types, for example the attribute 'mlsfilewrite' has the following 'types' mapped.

```
seinfo -amlsfilewrite -t | less
Types: 3158
   snort_exec_t
   audisp_var_run_t
   auditd_var_run_t
   comsat_var_run_t
   dbskkd_var_run_t
   ....
   ....
```

So therefore, using Boolean Rule number 4, we can imply that if the process has any of the above types of all the 3158 types which is mapped to attribute 'mlsfilewrite', then we can bypass the custom *-property (No write down and write equal rule) enforced by the SELinux security server.

It is in the hands of the Security Administrator to do label the process with this type so that (Hence we regard this as a trusted object) as this process labelled with '$t1$' will be able to write to all files at all levels on the SELinux MLS enabled system.

According to **Section 2.7.5**, if the Security Administrator wants to bypass custom *-property and wants to allow process to write to any file level, then it can be done as follows.

```
typeattribute mywritetype_t mlsfilewrite;
```

Where 'mywritetype_t' is the type of the process and hence once above is assigned, files of type 'mywritetype_t' will be able to write to files of all sensitivity levels (As per Section3.1 , $S0$ .... $S15.C0.c1023$)

3.6 SELinux policy Type enforcement and MLSConstrain evaluation order

The permission has to first be allowed by a TE allow rule, and then any constraints on the permission are evaluated and the permission is only allowed if all such constraints evaluate to true

Eg: Below shows all the 'allow' rules for the 'httpd_t' type

```
sesearch --allow --all -t httpd_log_t
/etc/selinux/mls/policy/policy.24 | grep -i httpd | less

   allow httpd_log_t httpd_log_t : filesystem associate ;

   allow httpd_suexec_t httpd_log_t : file { ioctl read create
getattr lock append open } ;

   allow httpd_php_t httpd_log_t : file { ioctl read getattr
lock append open } ;

   allow logwatch_t httpd_log_t : file { ioctl read getattr
lock open } ;

  ....

  ....
```

So therefore, if we need to run two services, if there's any information flow required between the subjects and the objects of the two respective services, then there should exist sufficient type enforcement rules inorder for the two services to operate as intended. So, provided these proper type enforcement rules are in place and then checks for constrains inorder to allow information flow against the subjects/objects (process, file, device, filesystem, etc) being interacted in any operation involved information flow (read/write/append/relabel, etc.)

However if there's no interaction between the subjects/objects of the two services, then even if such Type enforcement rules allow and then the mlsconstrain statements will allow or disallow the information flow (read/write/append/relabel, etc.) depending on the MLS levels of the interacted subjects/objects (process, file, device, filesystem, etc.)

3.7 Security Lattice for SELinux Type Enforcement & SELinux MLS

Flow of information through the lattice in an environment with only Type enforcement and MLS works in different ways as discussed in **Section 3.6**.

Below are some examples on how information flows in a lattice for:

An environment with only Type Enforcement (No Multi Level Security)

Assume we've a service in a SELinux system where it has the following characteristics and types:

```
service is started and executed using init program: init_t
type of executable for the service: service_exec_t
type for the service: service_t
type for logs directory for the service: service_log_t
type for configurations for the service to function:
service_conf_t
```

Following is a subset of generic dummy type enforcement rules allowed within the SELinux security policy for the service to function assuming the following:

- Able to start service using 'init'
- service is able to read its configuration file
- service is able to write logs to its corresponding directory

For simplicity to indicate the lattice information flow of type enforcement, as SELinux rules has a massive amount of Type enforcement rules according to **Section 2.7.9**, all allow rules are not included.

```
allow init_t  service_exec_t :  process execute;
allow service_exec_it  service_t : process transition;
allow service_t service_log_t: file dir write append;
allow service_t service_conf_t: file read;
```

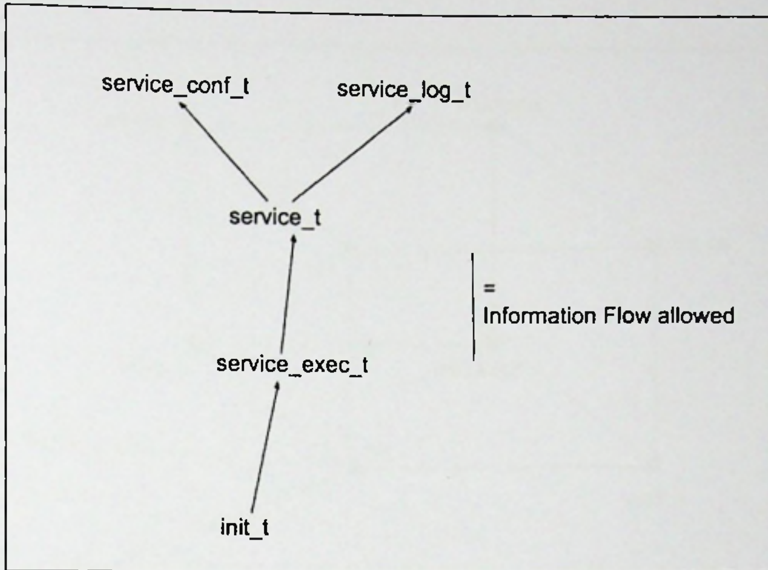


Figure 3.3 Lattice information flow for type enforcement rules

As discussed in **Section 3.6,** since this is not an MLS enabled environment and therefore all services are labelled with the same sensitivity ($s0$), therefore only type enforcement rules are evaluated by the SELinux security server and no mlsconstrains are evaluated (Simple property and customized *-Property are not evaluated).

An environment with SELinux Type Enforcement + Multi Level Security where:

Information flow between the types for the two services are allowed.
i.e: Type Enforcement rules already are in place

If type enforcement denies, SELinux security server denies the information flow and it doesn't reach to the point where mlsconstrain rules are evaluated. (i.e: Customized Bell Lapadula rules).

In a SELinux MLS enabled system, we know that the MLS dominance statements are such that: $S0 < S1$

Below shows a security lattice of information flow where $S0$ has $C0.C3$ categories and $S1$ has $C0.C1$ categories

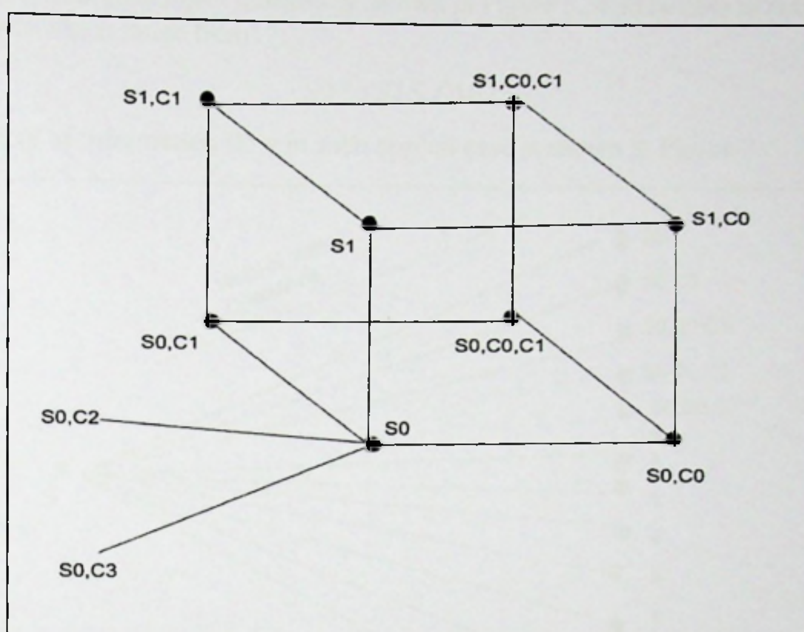


Figure 3.4 Lattice Information flow for Services running in labels S0.C0.C3 and S1.C0.C1

As per the deductions made in **Section 3.1,** we can deduce the following on Figure 3.4.

1. There's no information flow towards $S0$ from $S0.C2$ and $S0.C3$
2. $S0.C2$ can read information only from $S0$
3. $S0.C3$ can read information only from $S0$
4. No information flow can occur from $S1$ and $S0.C2$, $S0.C3$

An environment with special type mapped to an attribute in MLSConstrain statements

As discussed in **Section 3.5**, provided if Type enforcement rules permits information flow between the subjects and the objects and due to the nature of the mlsconstrain statements boolean checks, it is possible to allow information flow (Read or write depending on the mlsconstrain) provided the types for the subjects and the objects are assigned with the attributes allowed in the respective mlsconstrain statements for the simple property and custom *-property.

In this scenario, irrelevant of the security label assigned on the respective subjects and the objects, information will be allowed to either read or write, i.e: Bypassing the Bell lapadula security properties.

As computed in **Section 3.1,** there are 16 * 2^1024 possible labels as shown in Figure 3.5.

Therefore, a subject/object labelled $'S'$ shown in Figure 3.5will be able to read or write any labels which range from:

$$S0\ to\ S15.C1023$$

The lattice of information flow in such special case is shown in Figure 3.5.

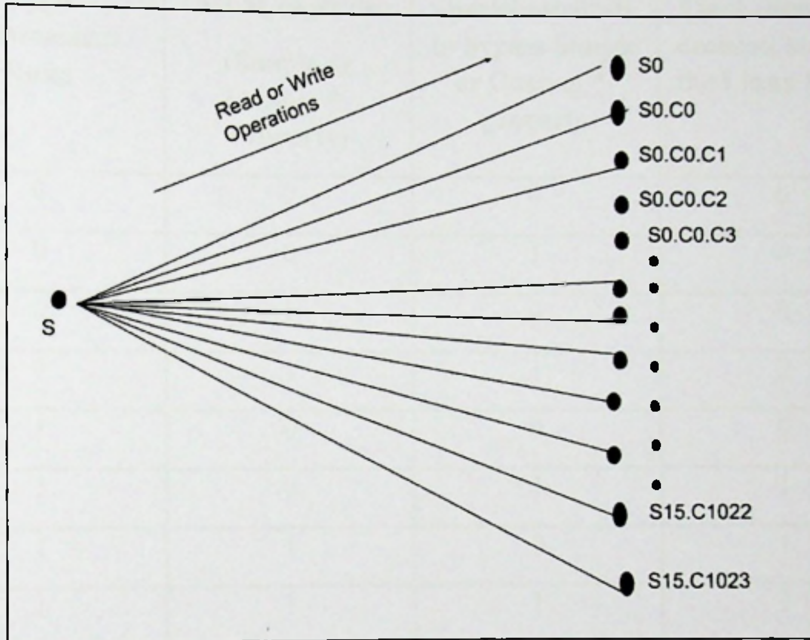


*Figure 3.5 Unrestricted Information Flow by bypassing MLSConstrain rules for Simple and custom *-Property*

3.8 Type Enforcement vs MLSConstrain vs Bypassing MLSConstrains

As discussed in **Sections 3.5, 3.6 and 3.7**, If Type enforcement disallows or there exist insufficient Type enforcement rules then the SELinux security server doesn`t go to the next step of the permission check which is checking on the boolean expressions specified in MLSConstrain statements.

If there exist sufficient Type enforcement rules, then the SELinux security server goes to the next step which is checking the Boolean expressions specified in MLSConstrain statements.

If Type enforcement and if any condition of MLSConstrain passes regardless of whether it is the simple property verification, custom *-property verification or the special attribute verification, then information flow is allowed.

So as per observation, there is a significant difference on how information flow is allowed or denied depending on how the below combination works as shown in Table 3.1.

In the Table 3.1below on the values '0' and '1'.

0 indicates that the Linux Kernel (Selinux Security Server) disallows it.

1 indicates that the Linux Kernel (Selinux Security Server) allows it.

Type Enforcement Rules	MLSConstrain (Simple or Custom *-Property)	Special attribute to bypass Simple or Custom *-property	Final allow/deny decision made by the Linux Kernel
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

*Table 3.1 Permutations of how the final decision is made upon the allowance and denials of Type Enforcement Rules, MLSConstrain rules and bypassing MLSConstrain Simple property / Custom *-Property rules*

3.9 Program to determine the Security levels for secure execution

To run multiple network services for secure execution in a SELinux MLS enabled system, the basis is, information flow shouldn't be allowed between the subjects and the objects of the respective services unless required. Inorder to make this happen, proper security labels should be assigned across the objects so that only the required information will flow through the security lattice for the services to be up & running.

Therefore as per **Section 3.5.1 and 3.5.2**, since already the SELinux security server enforces the custom Bell lapadula model using MLSConstrain statements on the subjects and the objects other than few exceptional cases which are attribute based, it is required to determine the security levels by value or at least in the form of an inequality of the network services which are intending to run on the SELinux MLS enabled system.

As discussed in **Section 2.4** it is still required to overcome the ambiguity of the security levels in the security lattice by evaluating conditions on how the subjects and the objects of the services running in the system interacts with each other.

UNIVERSITY OF MORATUWA
LIBRARY
SRI LANKA

For simplicity and deduction purposes, consider the SELinux MLS enabled system runs two network services. For example, there can be scenarios where the network services need to allow and share information across its corresponding related subjects and the objects and therefore the sensitivity levels or the labels in the lattice would change depending on this.

Thus, there can be several such scenarios and therefore the position of subjects/objects in the lattice would vary accordingly depending on how the information flow needs to occur for the two network services to be operational, up and running.

3.9.1 Decision making on determining security labels

Let's assume we've an SELinux MLS enabled System where the System has the default SELinux security contexts and modification to the security labels are done on the corresponding subjects and objects. i.e: Therefore, all security contexts obtained on the services components (i.e: subjects and the objects) will be of default security levels '$s0$'.

Therefore the "Sensitivity Level" of the two services will attain default SELinux sensitivities.

Eg.: The sensitivity level of '$svc1$' will be: $\boldsymbol{S0}$ and as well as for '$svc2$' will be: $\boldsymbol{S0}$

This SELinux MLS enabled system has the python program placed which has the capability to determine the security levels of the subjects and the objects of the services which needs to run depending on the requirement of the user (Determined and decided through input of the user parsed to the program).

The security levels are determined by utilizing the logic of lattice-based information flow as per the requirements entered by the user prompted by the program, this program is a python script.

For the purpose of running multiple network services and for simplicity we'll be aiming to run the two services on the system.

Let $S1$ and $S2$ br two services which runs in the SELinux MLS enabled system.

According to the SELinux Security Model, ie. Customized Bela-Lapadula model,

1. **Simple Property:** No read up (Read down allowed)
2. **Customized Star Property:** No write down and write equal

There can be several combinations of service $S1$ interacts and operates with service $S2$ as follows:

- $S1$ writes $S2$

- $S2$ writes $S1$

However, since the SELinux MLSconstraint as per **Section 3.5.1** allows write only to equal security levels, then if $S1$ can write to 2, then $S2$ imlicity can write to $S1$.

Let's denote this operation as $S1\ w\ S2$ where,

$$S1 = Sensitivity\ of\ service\ S1$$

$$S2 = Sensitivity\ of\ service\ S2$$

Similary using the simple property, service $S1$ and $S2$ can have the following combinations:

1. Service $S1$ needs to read objects of service $S2$.
   a. Let's denote this as $S1RS2$.
2. Service $S2$ need to read the contents of $S1$.
   a. Let's denote this as $S2RS1$.

A sensitivity label consists of two components.

$$L = \{S, C\}$$

Where L is the label, S is the sensitivity level and C is the category level which is on optional part of the label "L".

Therefore, when a Security administrator installs, configures and labels the subject/object of the services $S1$ and $S2$, the administrator has the option of assigning categories to the subject/object.

Therefore, it is required to determine the most secure labels. Ie. Determine the sensitivity levels and optionally choose categories for the respective service S1 and S2 according to the security lattice information flow.

Below is a table of various combinations of such scenarios, the table indicates mathematically how the sensitivities and categories should be choosen by the system administrator.

Category	S1 writes to S2	S1 read S2	S2 read S1	InEquality / Equation
0	0	0	0	$S1 \gtreqless S2$
0	0	0	1	$S1 > S2$
0	0	1	0	$S1 < S2$
0	0	1	1	$S1 = S2$
0	1	0	0	$If\ S1WS2 \rightarrow 1;\ then$ $S1 = S2$ $(BLP\ write\ equal\ property)$ $S1RS2 \rightarrow 1\ \&\ S2RS1 \rightarrow 1$ $\therefore skipping;$
0	1	0	1	$If\ S1WS2 \rightarrow 1;\ then$ $S1 = S2$ $(BLP\ write\ equal\ property)$ $S1RS2 \rightarrow 1\ \&\ S2RS1 \rightarrow 1$ $\therefore skipping;$
0	1	1	0	$If\ S1WS2 \rightarrow 1;\ then$ $S1 = S2$ $(BLP\ write\ equal\ property)$ $S1RS2 \rightarrow 1\ \&\ S2RS1 \rightarrow 1$ $\therefore skipping;$
0	1	1	1	$S1 = S2$
1	0	0	0	$C1 <> C2$ $S1 \gtreqless S2$
1	0	0	1	$S2 > S1$ $C1 \subseteq C2$
1	0	1	0	$S1 > S2$ $C2 \subseteq C1$
1	0	1	1	$S1 = S2$ $C1 \subseteq C2\ or\ C2 \subseteq C1$
1	1	0	0	$If\ S1WS2 \rightarrow 1;\ then$ $S1 = S2$ $(BLP\ write\ equal\ property)$ $S1RS2 \rightarrow 1\ \&\ S2RS1 \rightarrow 1$ $\therefore skipping;$

1	1	0	1	$If\ S1WS2 \rightarrow 1;\ then$ $S1 = S2$ $(BLP\ write\ equal\ property)$ $S1RS2 \rightarrow 1\ \&\ S2RS1 \rightarrow 1$ $\therefore skipping;$
1	1	1	0	$If\ S1WS2 \rightarrow 1;\ then$ $S1 = S2$ $(BLP\ write\ equal\ property)$ $S1RS2 \rightarrow 1\ \&\ S2RS1 \rightarrow 1$ $\therefore skipping;$
1	1	1	1	$L1 = L2$ $C1 \subseteq C2\ or\ C2 \subseteq C1$

Table 3.2 Inequality Table utilized by the Python script

3.9.2 Program Workflow

The program's workflow and assumptions will be as follows:

1. Capability to determine the security levels (Assign sensitivities and categories) of the subjects and the objects of the services which needs to run depending on the requirement of the user (Determined and decided through input of the user parsed to the program).
2. By default, all sensitivity levels of a SELinux MLS enabled system takes s0 which is the lowest secrecy level.
3. The user running this program (Shell script) should be aware and have the knowledge of any subject and object interaction (Read/write operations) which needs to occur for the two services to function and operate as per expectations.
4. The user has the ability to provide answers as 'inputs' to the program for the following scenarios of the 2 services.

To understand the flow of information between services $S1$ and $S2$, the ambiguity of the security levels of the respective services can be demonstrated by security lattice structures.

The flow of information with the lattice can be aligned with the customized Bell Lapadula security model (No read up, No write down and write equality) for various combinations shown in the Table 3.2($Category, S1WS2, S1RS2, S2RS1$).

1. $0000 \rightarrow S1 \gtreqless S2$

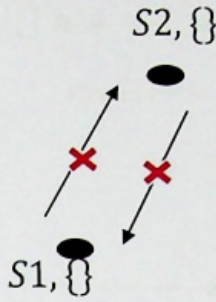


Figure 3.6 0000

2. $0001 \rightarrow S2 > S1$

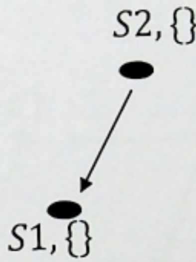


Figure 3.7 0001

3. $0010 \rightarrow S2 < S1$

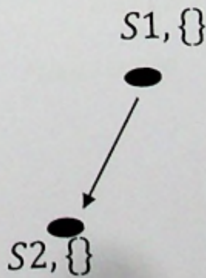


Figure 3.8 0010

UNIVERSITY OF MORATUWA LIBRARY

4. $0011 \rightarrow S2 = S1$

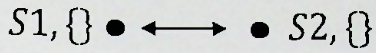


Figure 3.9 0011

5. $0111 \rightarrow S2 = S1$

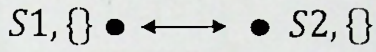


Figure 3.10 0111

6. $1000 \rightarrow C1 <> C2$
   $\rightarrow S1 \gtreqless S2$

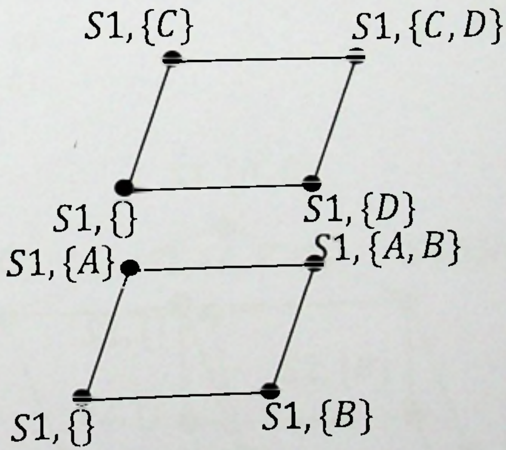


Figure 3.11 1000

7. $1001 \rightarrow S2 > S1$
$\rightarrow C1 \subseteq C2$

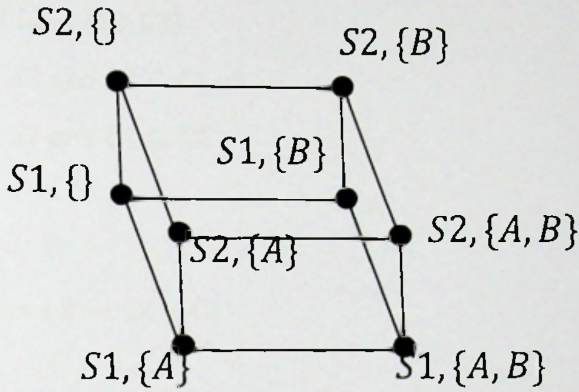


Figure 3.12 1001

8. $1010 \rightarrow S1 > S2$
$\rightarrow C2 \subseteq C1$

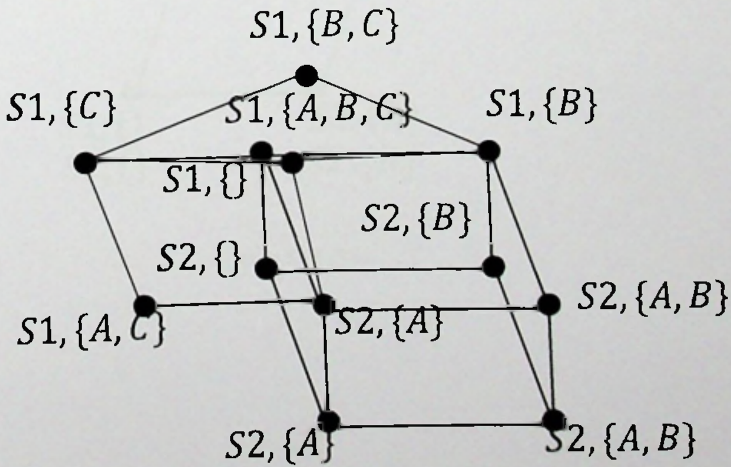


Figure 3.13 1010

9. 1011 $\rightarrow S1 = S2$
$\rightarrow C2 \subseteq C1$ *and* $C2 \subseteq C1$

As per **Section 2.4.2,**

$L1 = \{S1, C1\}$ *and* $L2 = \{S2, C2\}$

$L1 > L2$ *iff* $S1 > S2$ *and* $C2 \subseteq C1$ ***or***

$L1 < L2$ *iff* $S1 < S2$ *and* $C1 \subseteq C2$

However,

As $S1 = S2 \rightarrow L1 = L2 \rightarrow C2 = C1$

$\therefore$ *Lattice* $\rightarrow$

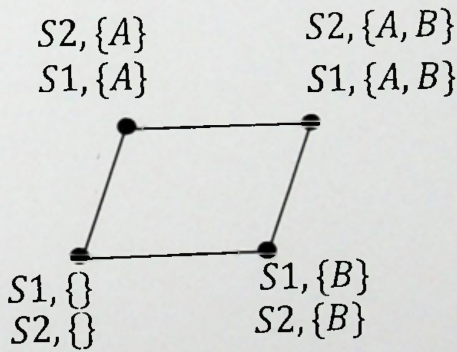


Figure 3.14 1011

10. 1111 $\rightarrow S1 = S2$

$\rightarrow C2 \subseteq C1 \; and \; C2 \subseteq C1$

As per **Section 2.4.2,**

$L1 = \{S1, C1\} \; and \; L2 = \{S2, C2\}$

$L1 > L2 \; iff \; S1 > S2 \; and \; C2 \subseteq C1 \; \boldsymbol{or}$

$L1 < L2 \; iff \; S1 < S2 \; and \; C1 \subseteq C2$

$However,$

$As \; S1 = S2 \; \rightarrow \mathrm{L1} = \mathrm{L2} \; \rightarrow \mathrm{C2} = \mathrm{C1}$

$\therefore \; Lattice \rightarrow$

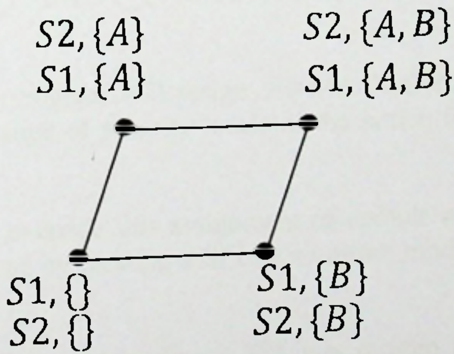


Figure 3.15 1111

3.10. Assigning the Security labels for secure execution

The program discussed in **Section 3.9** will output the security levels which needs to be assigned for the subjects and the objects in the form of an inequality, and if applicable any categories.

Network services in linux are started using startup scripts placed in /etc/init.d/ directory. The fiirst process which starts in a linux system with Process ID '1', which is also referred to as 'init' is responsible for starting any services placed in /etc/init.d directory provided they're enabled to start at system bootup.

However, there is a default security context assigned by the 'init' program when starting such services, the security context is hardcoded in the following file.

```
[root@msc-research-ishara-system1 /]# cat
/etc/selinux/mls/contexts/initrc_context

system_u:system_r:initrc_t:s0-s15:c0.c1023
```

Therefore, the init program will assign 'S0-S15.C0.C1023' to any program which covers the entire range of security levels of the lattice formed in a SELinux MLS system.

However, we can override this assignment of default security labels by the 'init' program, this is done by creating a SELinux custom module as discussed in **Section 2.6.11**.

Therefore, by creating and loading a SELinux custom module, we can assign the security levels determined by and outputted by the python program for the services intending to run on the SELinux MLS enabled system. The Security administrator can create these SELinux custom modules and then start the services to enforce the services to be running in the determined security levels outputted by the python program.

3.11 Generic SElinux policies for secure execution

Once the Security administrator assigns the security labels by creating the SELinux custom modules, but if there exist one or more boolean conditions which overrides the MLSConstrain for the simple property and custom *-property, then the purpose of assigning the security labels would be irrelevant as this will result in unrestricted information flow in the lattice and services will then be vulnerable to attack and non-secure. That is, information flow is allowed without the knowledge of the Security administrator to any label within the security lattice as discussed in **Section 3.7, figure 3.5.**

As per the comprehensive differences discussed in **Sections 3.7 and 3.8** on precedence of Type enforcement, enforcement of Custom Bell Lapadula model and overriding Custom Bell Lapadula by the SELinux security server, the following generic rules shown in Table 3.3 can be deduced to securely run network services in an SELinux MLS based system depending upon the security levels determined and outputted by the python program which can be followed by the Security administrator.

Therefore, as a rule of thumb, whenever a new service needs to be introduced to the SELinux MLS system, then the following generic checklist should be followed by the Security administrator in order to securely execute the service and the rest of the services on the system, thus restricting any unwanted information flow.

L1 and L2 denotes security levels outputted by the python program as indicated in Table 3.3.

InEquality produced by the python program	Type Enforcement Rule	MLSConstrain (Custom Bell-Lapadula Model)	Overriding MLSConstrain (Overriding Custom Bell-Lapadula Model)
$L1 \not\gtreqless L2$	If TE allow rules exist	Choose the Lower security of the two services to be of different levels Eg: $S0$ for $service1$ and $S2$ for $service2$, then information flow for writes will be denied. To deny information flow for 'reads', introduce categories.	Should ensure that the Boolean condition for the Bell lapadula policy is always satisfied, then the rest of the boolean expressions are ignored.
$L1 \not\gtreqless L2$	If o allow TE rules should exist i.e: Should Deny Information flow	Since TE denies information flow, this is irrelevant	Since TE denies information flow, this is irrelevant
L1 = L2	TE rules exist to allow Information flow	Should ensure that the Boolean condition for the Bell lapadula policy is always satisfied, then the rest of the boolean expressions in the mlsconstrain statements are ignored.	Should ensure that the Boolean condition for the Bell lapadula policy is always satisfied, then the rest of the boolean expressions are ignored.
L1 > L2	TE rules exist to allow Information flow	Should ensure that the Boolean condition for the Bell lapadula policy is always satisfied, then the rest of the boolean expressions in the mlsconstrain statements are ignored.	Should ensure that the Boolean condition for the Bell lapadula policy is always satisfied, then the rest of the boolean expressions are ignored.
L1 < L2	TE rules exist to allow Information flow	Should ensure that the Boolean condition for the Bell lapadula policy is always satisfied, then the rest of the boolean expressions in the mlsconstrain statements are ignored.	Should ensure that the Boolean condition for the Bell lapadula policy is always satisfied, then the rest of the boolean expressions are ignored.

Table 3.3 SELinux Generic Rule Precedence

4. SYSTEM/SOLUTION ARCHITECTURE AND IMPLEMENTATION

It is required to design, setup and implement the program as in **Section 3.9** to be run on a SELinux MLS enabled environment.

The rest of the sections in this chapter focuses on the design aspects of the program, then determining the security levels of the network services intending to run, setting up testbed environments composed with two SELinux MLS labelled environment and finally running the network services with the determined labels using the program

4.1. Interactive program to determine the security levels

Basically, the program's workflow will be as follows:

1. Has the capability to determine the security contexts (Determine sensitivities) of the subjects and the objects of the services which needs to run depending on the requirement of the user (Determined and decided through input of the user parsed to the program).
2. The user has the ability to provide answers as 'inputs' to the program for the following scenarios of the 2 services:
   a. The program prompt whether the '$svc1$' or '$svc2$' has any categories.
   b. The program prompts whether '$svc1$' needs to write to '$svc2$' and vice-versa.
   c. The program prompts whether the '$svc1$' needs to read content of '$svc2$' and vice-versa.
3. The program will evaluate the simple property and custom *-property and output the security levels in the form of an inequality for sensitivity levels and if applicable categories.

4.1.1. Pseudocode for the program

The pseudocode of the program shown below which is python based interactive program decides the security labels of the network services '$S1$' and '$S2$' intending to run on the SELinux MLS enabled system.

```
IF (S1 read S2) AND (S2 read S1)
    Print "S1 equals to S2"
        IF (has categories)
            Print "C1 is a subset of C2"
    EXIT 0
IF (S1 and S2 write to each other)
    Print "Skipping..."
ELSE
    IF (NOT S1 read S2) AND (NOT S2 read S1)
        Print "S1 and S2 are incomparable"
        IF (has categories)
            Print "C1 <> C2"
    EXIT 0
ELSE IF (S1 read S2) AND (NOT S2 read S1)
    Print "S1 > S2"
        IF (has categories)
            Print "C2 is a subset of C1"
    EXIT 0
ELSE IF (NOT S1 read S2) AND (S2 read S1)
    Print "S1 < S2"
        IF (has categories)
            Print "C1 is a subset of C2"
    EXIT 0
```

The python program is available in source code repository [32].

UNIVERSITY OF MORATUWA LIBRARY

4.2. Setting up SELinux MLS testbeds

It is required to setup two SELinux MLS enabled testbed environments which needs to be configured with appropriate MLS labels.

CentOS 6.10 will be installed in the Systems as virtual machines with the default SELinux MLS policies.

```
[root@msc-research-ishara-system1 ~]# yum install selinux-policy-mls -y

Loaded plugins: fastestmirror

Setting up Install Process

....

---> Package selinux-policy-mls.noarch 0:3.7.19-312.el6 will be installed
```

Note down that by default (By default SELinux mode will be in targeted mode), without MLS enabled, the sensitivity level assigned will '*s0*'.

```
SELinux status: enabled

SELinux mount: /selinux

Current mode: enforcing

Mode from config file: enforcing

Policy version: 24

Policy from config file: targeted
```

Enforce MLS mode using SELinux config file.

```
vi /etc/sysconfig/selinux

SELINUXTYPE=mls
```

Relabel the Filesystem

```
touch /.autorelabel && init 6
```

Snap of file system relabelling done by the SELinux security server.

```
Remounting root filesystem in read-write mode:              [      ]
Mounting local filesystems:                                  [      ]

*** Warning -- SELinux mls policy relabel is required.
*** Relabeling could take a very long time, depending on file
*** system size and speed of hard drives.
SC**********************_
```

Table 4.1 Filesystem file relabelling of a SELinux MLS enabled system

Check the available sensitivity levels available in the SELinux MLS system, there'll be 16 sensitivity levels in the following order of clearance.

$$s0 < s1 < s2 < s3 \ldots\ldots\ldots\ldots\ldots < s15$$

```
seinfo --sensitivity
Sensitivities: 16
   s0
   .....
   s16
```

Finally verify whether MLS is in enabled mode.

```
[root@msc-research-ishara-system1 /]# sestatus -v
SELinux status: enabled
SELinuxfs mount: /selinux
Current mode: enforcing
Mode from config file: enforcing
Policy version: 24
Policy from config file: mls
```

Repeated the same steps above to setup the second testbed environment (msc-research-ishara-system2) so that both environments will be identical.

4.3. Running the Program

There can be several scenarios where information flow is required or not between the two network services intending to run. Two such scenarios are considered which are:

1. *Service*1 needs to read/write content to & from *service*2.
2. No information flow should occur between *service*1 and *service*2.

The above two scenarios are tested against the two testbed environments created in **Section 4.2.**

Obtain the developed python program [32] which is available in github as follows:

```
git clone  https://github.com/ifernando/SELinux-MLS.git
```

Scenario 1
Service 1 needs to read/write content to and from Service 2.

```
python3 sensitivity-inequality.py
Does S1 and S2 have categories?(Y/N) : n
Can S1 and S2 write to each other?(Y/N) : y
Can S1 READ S2 content?(Y/N) : y
Can S2 READ S1 content?(Y/N) : y
--------------------------------------------------------------
---------
Your input is...
S1 READ S2 content :
YES
S2 READ S1 content :
YES
S1 WRITE S2 and S2 WRITE S1 content :
YES
Does S1 and S2 has categories :
NO
--------------------------------------------------------------
---------
```

```
Output is...
S1 = S2
```

As observed in the output above, sensitivity level of '*service*1' needs to be equal to the sensitivity level of '*service*2'.

Therefore, SELinux security labels assigned on the processes '*service*1' and '*service*2' should be the same and without any category component in the SELinux security context.

Scenario 2

Service 1 and Service 2 has no information flow.

```
python3 sensitivity-inequality.py
Does S1 and S2 have categories?(Y/N) : Y
Can S1 and S2 write to each other?(Y/N) : n
Can S1 READ S2 content?(Y/N) : n
Can S2 READ S1 content?(Y/N) : n
------------------------------------------------------------
---------
Your input is...
S1 READ S2 content :
NO
S2 READ S1 content :
NO
S1 WRITE S2 and S2 WRITE S1 content :
NO
Does S1 and S2 has categories :
YES
------------------------------------------------------------
---------
Output is...
```

S1 || S2

C1 <> C2

As observed in the output above, sensitivity level of '$service1$' can not be compared with '$service2$'.

The categories of $Service\ 1$ $(C1)$ and are not equal to categories $(C2)$ of $Service\ 2$.

Therefore SELinux security labels assigned on the processes for '$service1$' and '$service2$' is not the same (Hence no information flow will occur between the respective services), the labels will consist of both sensitivity levels and categories in the following form

$$service1 \rightarrow S1{:}\,C1.C3$$

$$service2 \rightarrow S2{:}\,C7.C9$$

4.4. Choosing the network services

To demonstrate on securely running the network services, two common widely used network services are chosen which would be Apache and MySQL. Apache webserver [33] is the most widely used webservers in the world and same goes with MySQL [34] which is a world's most popular relational database, both of them are open source as well.

4.4.1. SELinux module creation for Scenario 1

As per the observations in the scenario 1 and scenario 2, the sensitivity levels for services are the same in Scenario 1, however the sensitivity levels for the 2 services in scenario 2 is incomparable and no equal categories.

For **scenario 1**, information flow is required between $service1$ and $service2$ as indicated in **Section 4.3.1.**

The following is tested against the first testbed environment ($msc - research - ishara - system1$).

Lets $svc1$ be Apache web service and $svc2$ be MySQL service.

The inequality in **Section 4.3.2** indicated that:

Sensitivity level of Apache Service = Sensitivity level of MySQL Service.

Since we have no categorization to be added into the security levels of the respective services, we've the option of choosing sensitivity levels between $S0$ *to* $S15$ as per **Section 2.7.1.**

Sensitivity level of Apache Service = Sensitivity level of MySQL Service.

Let '$s0$: $s4$' be the sensitivity levels to be assigned to Apache and MySQL processes.

SELinux module for Apache process

Since apache is started by init, we need to determine & allow **init** to transition the processes for apache in which it is supposed to run which is determined as follows:

```
#Install the apache webserver package
yum install httpd -y


#Inspect the types of the objects interacted while starting apache


ls -lZ /usr/sbin/run_init
-rwxr-xr-x.    root    root    system_u:object_r:run_init_exec_t:s0
/usr/sbin/run_init


ls -lZ /usr/sbin/httpd
-rwxr-xr-x.    root    root    system_u:object_r:httpd_exec_t:s0
/usr/sbin/httpd
```

From following security context file for initrc process as run_init always tries to run the init script in the `initrc_t` context.

```
[root@msc-research-ishara-system1            mls]#            cat
/etc/selinux/mls/contexts/initrc_context
system_u:system_r:initrc_t:s0-s15:c0.c1023


Therefore we can deduce the following :
/usr/sbin/run_init process has type: initrc_t
/usr/sbin/httpd file executable has type: httpd_exec_t
/usr/sbin/httpd process has type: httpd_t

Type transition required : initrc_t ---> httpd_exec_t ---> httpd_t
Therefore, source type: initrc_t  and target type : httpd_t
```

Since the types for process transition are known now, the module can have created as follows:

Module for Apache

```
# cat > httpdtrans.te <<EOF
policy_module(httpdtrans, 1.0)

require {
	type initrc_t;
	type httpd_exec_t;
	type httpd_t;
}

range_transition initrc_t httpd_exec_t:process s0 - s4;

mls_rangetrans_source(initrc_t)
mls_rangetrans_target(httpd_t)
EOF
```

Compile the above range transition rule and install it on the SELinux MLS Policy so that it'll load it realtime.

```
make -f /usr/share/selinux/devel/Makefile httpdtrans.pp
semodule -i httpdtrans.pp
semodule  -l   |   grep -i httpd
```

SELinux Module for MySQL process

Like apache, MySQL Server is also started by **init**, we need to allow init to transition the processes for mysql in which it is supposed to run, it is determined as follows.

Install mysql client and mysql-server packages

```
yum  install mysql mysql-server


#Inspect the types of the objects interacted while starting MySQL


ls -lZ /usr/sbin/run_init
-rwxr-xr-x.    root    root    system_u:object_r:run_init_exec_t:s0
/usr/sbin/run_init


ls -lZ /usr/libexec/mysqld
-rwxr-xr-x.    root    root    system_u:object_r:mysqld_exec_t:s0
/usr/libexec/mysqld


 ls -lZ /usr/bin/mysqld_safe
-rwxr-xr-x.   root   root   system_u:object_r:mysqld_safe_exec_t:s0
/usr/bin/mysqld_safe


initrc process as run_init always run the init script in the initrc_t
context
```

Therefore we can deduce the following:

```
/usr/sbin/run_init process has type: initrc_t


/usr/libexec/mysqld file executable has type: mysqld_exec_t
/usr/libexec/mysqld process has type: mysqld_t


/usr/bin/mysqld_safe file executable has type: mysqld_safe_exec_t
/usr/bin/mysqld_safe process has type:  mysqld_safe_t


Type transition required for mysqld process : initrc_t
→  mysqld_exec_t →  mysqld_t
Therefore, source type: initrc_t  and target type : httpd_t
```

```
Type transition required for mysqld_safe process: initrc_t →
mysqld_safe_exec_t → mysqld_safe_t
```

To enforce above transitions and to label mysql related processes to $S0:S4$ (Same as the apache process).

Module for Mysqld process

```
# cat > mysqldtrans.te <<EOF
policy_module(mysqldtrans, 1.0)

require {
      type initrc_t;
      type mysqld_exec_t;
      type mysqld_t;
}

range_transition initrc_t mysqld_exec_t:process s0 - s4;

mls_rangetrans_source(initrc_t)
mls_rangetrans_target(mysqld_t)
EOF
```

Module for Mysqld_safe process

```
# cat > mysqld_safetrans.te <<EOF
policy_module(mysqld_safetrans, 1.0)

require {
      type initrc_t;
      type mysqld_safe_exec_t;
      type mysqld_safe_t;
}

range_transition initrc_t mysqld_safe_exec_t:process s0 - s4;
mls_rangetrans_source(initrc_t)
mls_rangetrans_target(mysqld_safe_t)
EOF
```

Compile the above range transition and sensitivity labelling rules and install it on the SELinux MLS Policy so that it'll load it realtime.

```
make -f /usr/share/selinux/devel/Makefile mysqldtrans.pp
make -f /usr/share/selinux/devel/Makefile mysqld_safetrans.pp
```

```
semodule -i mysqldtrans.pp
semodule -i mysqld_safetrans.pp
[root@msc-research-ishara-system1 /]# semodule   -l    |    grep -i
mysqld
mysqld_safetrans    1.0
mysqldtrans    1.0
```

Running the network services

Finally install start the network services to be run on the SELinux MLS enabled system.

```
[root@msc-research-ishara-system1 /]# run_init
/etc/init.d/httpd start
[root@msc-research-ishara-system1 /]# ps -eZ | grep -i httpd
system_u:system_r:httpd_t:s4     2096 ?     00:00:00 httpd
system_u:system_r:httpd_t:s4     2098 ?     00:00:00 httpd
```

As observed, apache processed is labelled & confined to run on '*s4*' sensitivity level.

Similarly start mysql server process.

```
[root@msc-research-ishara-system1 /]# run_init
/etc/init.d/mysqld start
[root@msc-research-ishara-system1 /]# ps -eZ | grep -i mysql
system_u:system_r:mysqld_safe_t:s3 3120 pts/0  00:00:00
mysqld_safe
system_u:system_r:mysqld_t:s3   3222 pts/0 00:00:00 mysqld
```

As observed, the mysql-server related processes which are '`mysqld`' and '`mysqld_safe`' are labelled & confined to run on sensitivity '*s3*'.

4.4.2. SELinux module creation for Scenario 2

As per the observations for scenario2, the sensitivity levels for the 2 services in scenario 2 is incomparable and no equal categories as per **Section 4.3.1.**

The following is tested against the first testbed environment (`msc-research-ishara-system2`).

Lets $svc1$ be Apache web service and $svc2$ be MySQL service.

The inequality in **Section 4.3.2** indicated that:

Sensitivity level of Apache Service is incomparable with the Sensitivity level of MySQL Service.

The categories of $Service\ 1\ (C1)$ and are not equal to categories of $Service\ 2\ (C2)$.

Therefore SELinux security labels assigned on the processes for '$service1$' and '$service2$' is not the same (Hence no information flow will occur between the respective services), the labels will consist of both sensitivity levels and categories in the following form

Since we have need to enabled categorization into the security levels of the respective services, we've the option of choosing sensitivity levels between $S0.C0\ to\ S15.C1023$.

Let $S0 - S4{:}\ C0.C2$ be the security label for Apache process.

$S1 - S2{:}\ C3.C5$ be the security label for Mysql process.

The testbeds are identical, and the services are started by the 'init' program, therefore the same type of transition rules as explained in **Section 4.4.1** will be created for **Scenario 2** as well, except the security labels will change as follows:

UNIVERSITY OF MORATUWA LIBRARY SRI LANKA

For Apache process

```
# cat > httpdtrans_with_cat.te <<EOF
policy_module(httpdtrans_with_cat, 1.0)

require {
      type initrc_t;
      type httpd_exec_t;
      type httpd_t;
}

range_transition initrc_t httpd_exec_t:process s0 - s4:c0,c1,c2;

mls_rangetrans_source(initrc_t)
mls_rangetrans_target(httpd_t)
EOF
```

Compile the above range transition rule and install it on the SELinux MLS Policy so that it'll load it realtime.

```
make -f /usr/share/selinux/devel/Makefile httpdtrans_with_cat.pp
semodule -i httpdtrans_with_cat.pp
semodule  -l   |   grep -i httpd
```

For MySQL

```
# cat > mysqldtrans_with_cat.te <<EOF
policy_module(mysqldtrans_with_cat, 1.0)

require {
      type initrc_t;
      type mysqld_exec_t;
      type mysqld_t;
}

range_transition initrc_t mysqld_exec_t:process s1 - s2:c3,c4,c5;

mls_rangetrans_source(initrc_t)
mls_rangetrans_target(mysqld_t)
EOF
```

Module for Mysqld_safe process

```
# cat > mysqld_safetrans_with_cat.te <<EOF
policy_module(mysqld_safetrans_with_cat, 1.0)

require {
      type initrc_t;
      type mysqld_safe_exec_t;
      type mysqld_safe_t;
}

range_transition initrc_t mysqld_safe_exec_t:process  s1 -
s2:c3,c4,c5;

mls_rangetrans_source(initrc_t)
mls_rangetrans_target(mysqld_safe_t)
EOF
```

Compile the above range transition and sensitivity labelling rules and install it on the SELinux MLS Policy so that it'll load it realtime.

```
make -f /usr/share/selinux/devel/Makefile mysqldtrans_with_cat.pp
make              -f             /usr/share/selinux/devel/Makefile
mysqld_safetrans_with_cat.pp

semodule -i mysqldtrans_with_cat.pp
semodule -i mysqld_safetrans_with_cat.pp

[root@msc-research-ishara-system2 /]# semodule   -l    |    grep -i
mysqld
mysqld_safetrans_with_cat    1.0
mysqldtrans_with_cat    1.0
```

Running the network services

Finally install start the network services to be run on the SELinux MLS enabled system.

```
[root@msc-research-ishara-system2 /]# run_init
/etc/init.d/httpd start

[root@msc-research-ishara-system2 /]# ps -eZ | grep -i http

system_u:system_r:httpd_t:s0-s4:c0.c2 17968 ?  00:00:00 httpd

system_u:system_r:httpd_t:s0-s4:c0.c2 17970 ?  00:00:00 httpd
```

As observed, apache processed is labelled & confined to run on '$s0 - s4{:}\, c0.\, c2$' security level.

Similarly start mysql server process

```
[root@msc-research-ishara-system2 /]# run_init
/etc/init.d/mysqld start

[root@msc-research-ishara-system2 /]# ps -eZ | grep -i mysql

system_u:system_r:mysqld_safe_t:s1-s2:c3.c5 19395 pts/0
00:00:00 mysqld_safe

system_u:system_r:mysqld_t:s1-s2:c3.c5 19497 pts/0 00:00:00
mysqld
```

As observed, the mysql-server related processes which are 'mysqld' and 'mysqld_safe' are labelled & confined to run on sensitivity '$s1 - s2{:}\, c3.\, c5$'

5. System Evaluation and Analysis

In the previous Chapter, two scenarios were evaluated where information flow is allowed from Apache process to Mysql process and information flow to and from Apache process to Mysql process is not allowed.

To verify the Scenario 1 in Chapter 4, i.e: Allow information flow from apache process to read and write content to and from Mysql, a real-world application can be installed.

Wordpress [35] application running on the SELinux MLS enabled system would be a good option to test **Scenario 1** as wordpress heavily relies on Apache which is the frontend and MySQL as the backend.

For wordpress to function, information flow between Apache and MySQL should be allowed.

5.1 Analysis of testbed 1 environment

Below steps show on setting up wordpress on the 1st test bed environment (`msc-research-ishara-system1`) where both Apache and MySQL is confined to run on sensitivity levels $S0 - S4$.

```
#Install php and php-mysql related packages
yum install php php-mysql
yum install php-mysql php-pdo php-pear php-pecl php-xml php-gd php-
zlib


#Download and install wordpress
wget http://wordpress.org/latest.tar.gz
tar -xzvf latest.tar.gz


#Create database in MySQL server for wordpress
mysql -u root -p


mysql> create database wordpress;
Query OK, 1 row affected (0.00 sec)


mysql> create user ishara@localhost;
Query OK, 0 rows affected (0.00 sec)


mysql> set password for ishara@localhost=password("selinux");
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> grant all privileges on wordpress.* to ishara@localhost
identified by 'selinux';
Query OK, 0 rows affected (0.00 sec)

mysql> flush privileges
     -> ;
Query OK, 0 rows affected (0.00 sec)

exit ;

#Copy wordpress configuration and configure mysql configs for
wordpress
cp ~/wordpress/wp-config-sample.php ~/wordpress/wp-config.php

vi ~/wordpress/wp-config.php

// ** MySQL settings - You can get this info from your web host **
//
/** The name of the database for WordPress */
define('DB_NAME', 'wordpress');

/** MySQL database username */
define('DB_USER', 'ishara');

/** MySQL database password */
define('DB_PASSWORD', 'selinux');

/** MySQL hostname */
define('DB_HOST', 'localhost');

cp -r ~/wordpress/* /var/www/html

#Finally restart apache to pick wordpress php related files
run_init  /etc/init.d/httpd restart
```

Wordpress is accessible without any issue as information flow is allowed between Apache and MySQL.

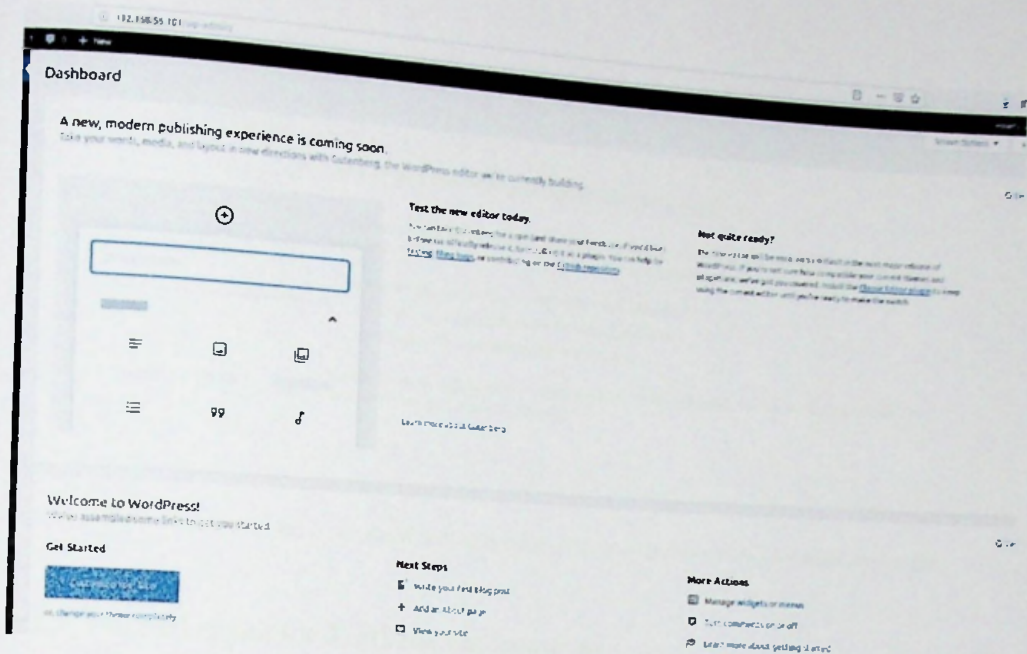


Figure 5.1 Wordpress is up and running

5.1.1 State Diagram for Testbed1 environment

Refer to Appendix B

5.2 Analysis of testbed 2 environment

In the 2nd test bed environment (`msc-research-ishara-system2`) where Apache is labelled with $S0 - S4{:}\ C0.C2$ and MySQL is labelled with $s1 - s2{:}\ c3.c5$.

However unlike in testbed1, in testbed2, the following errors throw up when accessing wordpress. The reason is the information between Apache and MySQL is full restricted, thus no information is allowed at all.

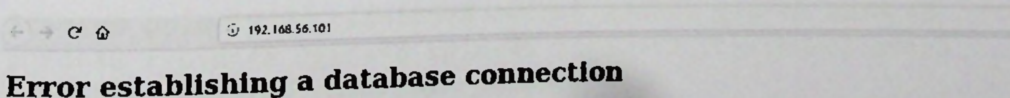


Figure 5.2 Wordpress is not functional as apache process is unable to write to mysql socket file

192.168.56.101/wp-admin/install.php

Error establishing a database connection

This either means that the username and password information in your wp-config.php file is incorrect or we can't contact the database server at localhost. This could mean your host's database server is down.

- Are you sure you have the correct username and password?
- Are you sure that you have typed the correct hostname?
- Are you sure that the database server is running?

If you're unsure what these terms mean you should probably contact your host. If you still need help you can always visit the WordPress Support Forums.

Figure 5.3 Wordpress is not functional as apache process is unable to write to mysql socket file

5.2.1 State Diagram for Testbed2 environment

Refer to Appendix C

5.2.2 Problem Analysis in Testbed2 environmet

To diagnose and dig into the details of this issue, as per **Section 2.6.3**, the SELinux Audit logs can be viewed as follows:

```
[root@msc-research-ishara-system2 ~]# ausearch -i -m AVC -ts recent

type=AVC msg=audit(1542126727.351:13074352): avc:  denied  { write } for  pid=19600 comm="httpd" name="mysql.sock" dev=dm-0 ino=787766 scontext=system_u:system_r:httpd_t:s0-s4:c0.c2 tcontext=system_u:object_r:mysqld_var_run_t:s1

type=SYSCALL msg=audit(1542126727.351:13074352): arch=c000003e syscall=42 success=no exit=-13 a0=b a1=7fff957a2540 a2=6e a3=0 items=0 ppid=19594 pid=19600 auid=0 uid=48 gid=48 euid=48 suid=48 fsuid=48 egid=48 sgid=48 fsgid=48 tty=(none) ses=3 comm="httpd" exe="/usr/sbin/httpd" subj=system_u:system_r:httpd_t:s0-s4:c0.c2 key=(null)

type=AVC msg=audit(1542126727.351:13074353): avc:  denied  { write } for  pid=19600 comm="httpd" name="mysql.sock" dev=dm-0 ino=787766 scontext=system_u:system_r:httpd_t:s0-s4:c0.c2 tcontext=system_u:object_r:mysqld_var_run_t:s1
```

```
type=SYSCALL msg=audit(1542126727.351:13074353): arch=c000003e
syscall=42 success=no exit=-13 a0=b a1=7fff957a25c0 a2=6e a3=0
items=0 ppid=19594 pid=19600 auid=0 uid=48 gid=48 euid=48
suid=48 fsuid=48 egid=48 sgid=48 fsgid=48 tty=(none) ses=3
comm="httpd" exe="/usr/sbin/httpd"
subj=system_u:system_r:httpd_t:s0-s4:c0.c2 key=(null)
```

To verify whether the the above violation is due to a Type enforcement rule or due to an mlsconstrain, try creating a custom SELinux type enforcement rule using audit2allow command.

Paste the above denied contents of logs into a temporary file (eg: /tmp/mysqlsock)

```
cat /tmp/mysqlsock

type=AVC msg=audit(1542126727.351:13074352): avc:  denied  {
write } for  pid=19600 comm="httpd" name="mysql.sock" dev=dm-0
ino=787766 scontext=system_u:system_r:httpd_t:s0-s4:c0.c2
tcontext=system_u:object_r:mysqld_var_run_t:s1
tclass=sock_file

type=SYSCALL msg=audit(1542126727.351:13074352): arch=c000003e
syscall=42 success=no exit=-13 a0=b a1=7fff957a2540 a2=6e a3=0
items=0 ppid=19594 pid=19600 auid=0 uid=48 gid=48 euid=48
suid=48 fsuid=48 egid=48 sgid=48 fsgid=48 tty=(none) ses=3
comm="httpd" exe="/usr/sbin/httpd"
subj=system_u:system_r:httpd_t:s0-s4:c0.c2 key=(null)
```

Create SELinux type enforcement rule using audit2allow.

```
cat /tmp/mysqlsock | audit2allow -M httpdtomysqlsock

#This will create a .te file called 'httpdtomysqlsock.te',
inspect this file

cat httpdtomysqlsock.te

module mysqlsock 1.0;

require {

    type httpd_t;

    type mysqld_var_run_t;

    class sock_file write;

}

#============= httpd_t ==============

#!!!! This avc is a constraint violation.  You will need to add
an attribute to either the source or target type to make it
work.

#Contraint rule:

allow httpd_t mysqld_var_run_t:sock_file write;
```

Now install the module

```
semodule -i  httpdtomysqlsock.pp
```

Even after the above module is installed, the same error shows up in the SELinux audit logs

This indicates that the errors are NOT due to any Type enforcement violations but violations due to mlsconstrain

Also as per obervation, the highlighted lines in the **audit logs** indicates that Bell Lapadula custom *-property (no write down, write equal property) is applied using the mlsconstrain statements.

From **Section 3.5.2**, below is the custom *-property corresponding to the Bell Lapadula model enforced as an mlsconstrain.

```
# the "single level" file "write" ops
mlsconstrain { file } { write create setattr relabelfrom append
unlink link rename }
        (( l1 eq l2 ) or
        (( t1 == mlsfilewritetoclr ) and ( h1 dom l2 ) and ( l1
domby l2 )) or
        (( t2 == mlsfilewriteinrange ) and ( l1 dom l2 ) and ( h1
domby h2 )) or
        ( t1 == mlsfilewrite ) or
        ( t2 == mlstrustedobject ));
```

The above multi level security constrain states that:
File `read/getattr/execute` permission are only allowed if:

1. The process low-level ($l1$) is equal to the file low-level ($l2$).

   Or

2. The process type ($t1$) has the mlsfilewritetoclr (write-up-to-clearance) Attribute and the process high-level ($h1$) dominates the file low-level ($l2$) and the process low-level ($l1$) is dominated by the file low-level ($l2$).

   Or

3. The process type ($t2$) has the mlsfilewriteinrange attribute and process low-level (l1) dominates file low-level ($l2$) and process high-level ($h1$) is dominated by file high-level ($h2$).

   Or

4. The process type ($t1$) has the mlswrite attribute.

   Or

5. The file type ($t2$) has the mlstrustedobject (e.g. `/dev/null`) attribute.

As shown in the SELinux Audit logs and the state diagram shown in **Appendix C**:

httpd process which is labelled with type: 'httpd_t' which has a security label of $S0 - S4{:}C0.C2$.

`scontext=system_u:system_r:httpd_t:s0-s4:c0.c2`

Similarly, the security labels assigned on the mysql socket file:

`tcontext=system_u:object_r:mysqld_var_run_t:s1`

According to the first boolean condition indicated above, the value of l1 and l2 are as follows

`l0 :  S0`

`l1 : S1`

Since `10 ≠ 11`, then the first boolean condition itself fails

If the rest of the conditions are looked, still the types óf scontext and tcontext which are **httpd_t** and **mysqld_var_run_t** does not belong to any of the 4 attributes shown below:

```
mlsfilewritetoclr

mlsfilewriteinrange

mlswrite

mlstrustedobject
```

Therefore, hence we can conclude as per the **Table 3.3** indicated in **Section 3.11**, though type enforcement rules for information flow between Apache and MySQL is in place, then if any of the boolean conditions in mlsconstrain fails, then information flow isn't allowed. Thus, in Scenario2 which interprets why Wordpress isn't functional

5.3 Verifying the generic SELinux Rules

Therefore, as per Table 3.3 in Section 3.11, we've identified the following two scenarios and following summarizes the generic rules which should be enforced by the Security Administrator for secure information flow within the SELinux Multi level security lattice.

Scenario1

$L1 = L2$ as outputs by the python script where information flow (read/write) is allowed between Apache and MySQL.

Type enforcement rules should exist to allow information flow between the object types for Apache and MySQL.

Sensitivity level S0:S4 were assigned for both Apache and MySQL processes as per output of python script ($S1 = S2$).

For wordpress to function, as determined in **Section 5.2.1 and 5.2.2**, there'll be read/writes between the corresponding subjects and objects for Apache and MySQL, the above labels satisfied for proper functionality of wordpress web application.

Scenario2

L1 $\gtrless$ L2 as outputs by the python script where no information flow (read/write) was to be allowed between Apache and MySQL

Type enforcement rules existed to allow information flow between the object types for Apache and MySQL as testbed1 and testbed2 environments were similar in nature

Since TE rules existed which allows information flow between Apache and MySQL, categorization was introduced as per **Table 3.3** in **Section 3.11**

Sensitivity levels S0-S4:C0.C2 & S1-S2:C3.C5 were assigned for both Apache and MySQL processes respectively as per output of python script (S1 $\gtrless$ S2, C1 <> C2)

For wordpress to function, as determined in **Section 5.2.1 and 5.2.2,** there'll be read/writes between the corresponding subjects and objects for Apache and MySQL, the above labels did not satisfy for proper functionality of wordpress web application as per the analysis done in **Section 5.2.2**

6. CONCLUSION

SELinux MLS security policies uses a modified version of the classic Bell-Lapadula model which fully takes care of information leak (*-property) thus providing confidentiality and integrity achieved via prohibiting "write up". That is, the subjects will be able to write to objects of the same sensitivity level whereas in the traditional Bell Lapadula model the subjects could write to objects which are of equal or more sensitivity which breaks the integrity protocol.

SELinux MLS overcomes the limitations of the traditional MLS model where the information flow in the traditional MLS lattice is too static, the decisions on to allow or deny information flow are solely based upon on the security level state of a given object. However SELinux MLS adds more fine grained rules where type enforcement rules plays a key part of allowing information flow in the MLS lattice. It was also found that the Bell-Lapadula model security policies could be easily bypassed to allow unrestriced information flow in the security lattice irrelevant of the security state labels assigned on a given subject or object. Therefore a Security Administrator should be aware and choose the security labels and the types for the processes so that information flows are allowed only if required.

By default, an SELinux MLS enabled system assigns the 's0' sensitivity level to a majority of the objects except kernel related objects. The subjects spawned by the init program by default attains the S0-S15.C0.1023 composing of all the possible label combinations in the security lattice. Therefore if one service is compromised, then the other services will be at risk of getting compromised provided if SELinux type enforcement rules & the multi level security constraints are applied incorrectly. There are numerous amount of ways to bypass the bell lapadula information flow model due to the nature of boolean conditional expressions present in the current implementation of SELinux policies.

One major possibility of a network service flaw would be information leak due to information flow from one service to another or a transitive information flow across the system subjects/objects finally to another network service causing the data to be leaked to the outside world. So to address this concern, the approach taken in this thesis is to prohibit information flow in addition to the mandatory access controls already facilitated by the SELinux security server in the form of a generic rule set to be followed by the Security Administrator.

The python script has the ability to decide and show the sensitivity levels for ambiguous sensitivities assigned for 2 network services in an SELinux MLS enabled system, however the script needs to be improved so that it will automatically apply the determined sensitivity levels to the corresponding subjects and objects in the SELinux enabled system without the need for the Security Administrator to apply these security levels as labels on the subjects & the objects manually.

Use of Discretionary control is still the majority of the systems use which are very vulnerable to attack, this is due to the fact that DAC does not protect against malicious code, Trojans and also the Owner of a subject/object in a DAC based system has the freedom to do anything with it while also the owner of a subject/object inherits all the privileges associated with that user.

RBAC concept is used very little in SELinux where as TE is heavily used more than RBAC, works such that every element in the system has a security type and has an access control matrix upon the type of subject (Which has a type too) which accesses an object and for MLS, every element in the system has an "MLS Level" or "Security Clearance" which is based on the Bell-Lapadula Model. For every security operation on the system, a set of MLS constraints are checked which is already embedded in the SELinux security server with the MLS Level of the subject and the object.

The default SELinux policy rules (without enabling MLS policy in SELinux) mostly addresses on the TE policies. The SELinux policy version used for the analysis was version no: 24.0, this consisted of more than 400000 TE rules which includes 3158 object types, 81 classes, 14 Roles and 27 initial SID's causing it impractical for manual analysis and inspection.

Due to flask architecture of SELinux, we can write our own policy and can be loaded into the security server thereby overcoming any limitations imposed by the current SELinux policy architecture. Writing such policies from scratch can be a very complex & time-consuming task and therefore the approach taken in this thesis was to utilize the current implementation of SELinux Mandatory access control with MLS Policies, type enforcement rules thereby using the lattice-based information flow approach to minimize unwanted information flow unless explicitly specified and allowed by the Security Administrator.

REFERENCES

[1] Loscocco, Peter, and Stephen Smalley, "Meeting critical security objectives with security-enhanced linux.," *Proceedings of the 2001 Ottawa Linux symposium*, pp. 115-134, 2001.

[2] Ahn, Gail-Joon, Wenjuan Xu, and Xinwen Zhang, "Systematic policy analysis for high-assurance services in SELinux," *InPolicies for Distributed Systems and Networks, 2008. POLICY 2008*, pp. 3-10, 2008.

[3] Hicks, Boniface, Sandra Rueda, Luke St Clair, Trent Jaeger, and Patrick McDaniel, "A logical specification and analysis for SELinux MLS policy," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 3, p. 26, 2010.

[4] B. a. S. D. S. Sarna-Starosta, "Policy analysis for security-enhanced linux," in *Proceedings of the 2004 Workshop on Issues in the Theory of Security (WITS)*, 2004.

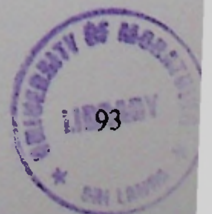
[5] S. C. V. a. W. S. Smalley, "Implementing SELinux as a Linux security module," NAI Labs Report 1.43.

[6] S. Q. Blake, "The clark-wilson security model," Indiana University of Pennsylvania, Library Resources, 2000. [Online]. Available: Retrieved from the World Wide Web at http://www. lib. iup. edu/comscisec/SANSpapers/blake. htm. [Accessed 10 January 2009].

[7] N. a. I. T. Balon, "The Biba Security Model," 2004.

[8] "Security-Enhanced Linux," NSA, [Online]. Available: https://www.nsa.gov/what-we-do/research/selinux/faqs.shtml#I1. [Accessed 10 11 2018].

[9] Hicks, Boniface, Sandra Julieta Rueda, Trent Jaeger, and Patrick D. McDaniel, "From Trusted to Secure: Building and Executing Applications That Enforce System Security," *USENIX Annual Technical Conference*, vol. 7, p. 34, 2007.



[10] A. Y. F. a. Y. E. Shabtai, "Securing Android-powered mobile devices using SELinux," *IEEE Security & Privacy 8*, vol. 8, no. 3, pp. 36-44, 2010.

[11] S. a. T. F. Smalley, "A security policy configuration for the Security-Enhanced Linux," NAI Labs Technical Report, 2001.

[12] S. J. M. Demurjian, "Implementation of mandatory access control in role-based security system," CSE367 Final Project report , 2001.

[13] N. I. T. Balon, "Biba security model comparison," in *Biba Security Model Comparison*, 2004.

[14] R. S. Sandhu, "Lattice-based access control models," *Computer* , vol. 26, no. 11, pp. 9-19, 1993.

[15] S. S. P. L. H. D. A. J. L. Ray Spencer, "The flask security architecture: system support for diverse security policies," in *8th USENIX Security Symposium*, Washington, 1999.

[16] F. K. M. a. D. C. Mayer, "Open Source Software Development Series," in *SELinux by Example*, Prentice Hall, 2007.

[17] B. McCarty, Selinux: Nsa's open source security enhanced linux, vol. 238, O'Reilly, 2005.

[18] R. Haines, The SELinux Notebook, Volume 1, The Foundations, 2nd Edition, 2010.

[19] "Red Hat Enterprise Linux 4: Red Hat SELinux Guide," RedHat, [Online]. Available: https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/4/html/SELinux_Guide/selg-chapter-0013.html. [Accessed 11 11 2018].

[20] "SELinuxProject/refpolicy," [Online]. Available: https://github.com/SELinuxProject/refpolicy. [Accessed 11 11 2011].

[21] "NB PolicyType," 13 September 2010. [Online]. Available: http://selinuxproject.org/page/NB_PolicyType. [Accessed 11 November 2011].

[22] D. Walsh, "Dan Walsh's Blog," 19 February 2009 . [Online]. Available: https://danwalsh.livejournal.com/26759.html. [Accessed 11 11 2018].

[23] "TresysTechnology/refpolicy," 23 June 2018. [Online]. Available: https://github.com/TresysTechnology/refpolicy/wiki. [Accessed 11 November 2018].

[24] D. E. L. J. L. Bell, Secure computer systems: Mathematical foundations, No. MTR-2547-VOL-1, MITRE CORP BEDFORD MA, 1973.

[25] K. J. Biba, Integrity considerations for secure computer systems. No. MTR-3153-REV-1, MITRE CORP BEDFORD MA, 1975.

[26] X. Xu, "A study on confidentiality and integrity protection of SELinux," in *Networking and Information Technology (ICNIT), 2010 International Conference*, 2010.

[27] "ConstraintStatements," 30 November 2009. [Online]. Available: http://selinuxproject.org/page/ConstraintStatements. [Accessed 11 11 2018].

[28] C. Hanson, "Selinux and mls: Putting the pieces together," in *Proceedings of the 2nd Annual SELinux Symposium*, 2006.

[29] C. PeBenito, "TresysTechnology/setools3," 4 May 2016. [Online]. Available: http://oss.tresys.com/projects/setools. [Accessed 11 11 2018].

[30] T. R. S. a. X. Z. Jaeger, "Analyzing integrity protection in the SELinux example policy," in *Proceedings of the 12th conference on USENIX Security Symposium-Volume 12. USENIX Association*, 2003.

[31] Y. Y. S. a. T. T. Nakamura, "SEEdit: SELinux Security Policy Configuration System with Higher Level," in *Proceedings of LISA '09: 23rd Large Installation System Administration Conference*, 2009.

[32] M. Fernando, Github, 11 11 2018. [Online]. Available: https://github.com/ifernando/SELinux-MLS/blob/master/sensitivity-inequality.py. [Accessed 11 11 2018].

[33] "Apache HTTP Server Project," Apache, [Online]. Available: https://httpd.apache.org/. [Accessed 13 11 2011].

[34] MySQL, 06 10 2018. [Online]. Available: https://github.com/mysql/mysql-server. [Accessed 11 11 2018].

[35] Wordpress, 06 10 2018. [Online]. Available: https://github.com/WordPress/WordPress. [Accessed 12 11 2018].

[36] J. D. A. L. H. J. D. R. a. C. W. S. Guttman, "Verifying information flow goals in security-enhanced Linux," *Journal of Computer Security 13,* no. 1, pp. 115-134, 2005.

APPENDICES

Appendix A – Comparison of SELinux Analysis tools

Analysis tool	Safety analysis	Completeness Analysis	Integrity Analysis	SOD Analysis	Information Flow Analysis	Method of Analysis	Policy Browsing	Rewriting the Policy	Method of modeling	Query Language	Macro Expansion
APOL	x		x	x	x	Informati on Flow	x		Syntactic Analysis	Selecting Attributes from menus	x
GOKYO	x	x	x			AC spaces-TCB			AC spaces-Graphical AC model (Sets)	x	x
SEEDIT	x			x	x	Higher level language, SPDL		x	Grouping Permissio ns-Access log user decision		

Appendix B – State Diagram for Testbed1 environment to represent information flow where wordpress application to function

Legends

Information flow allowed

Information flow Not allowed

S0 /var/www/html

Read/Write

Spawns Process

httpd_exec_t S0

httpd_t S0-S4

Execute

Process Transition

Write

initrc_t S0-S4

/var/lib/myql/mysql.sock S0

S0-S4 mysqld_t

Execute

Read/Write

mysqld_exec_t S0

Spawns Process

/var/lib/mysql/* S0

Appendix C – State Diagram for Testbed2 environment to represent information flow where wordpress application malfunctions