Comenius University in Bratislava
Faculty Of Mathematics, Physics And Informatics

Using SELinux to Enforce Two-Dimensional Labelled Security Model with Partially Trusted Subjects

MASTER'S THESIS

Study programme: Computer Science
Field of Study: 2508 Computer Science, Informatics
Department: Department of Computer Science
Supervisor: RNDr. Jaroslav Janáček, PhD.

Bratislava 2012 Bc. Martin Jurčík

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Martin Jurčík
Študijný program: informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: 9.2.1. informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Použitie SELinux-u na realizáciu dvojdimenzionálneho značkovaného bezpečnostného modelu s čiastočne dôveryhodnými subjektami

Cieľ: Cieľom tejto práce je implementovať náš dvojdimenzionálny značkovaný bezpečnostný model s čiastočne dôveryhodnými subjektami v operačnom systéme Linux využitím mechanizmu SELinux. Od autora práce sa očakáva vylepšenie implementácie prezentovanej v bakalárskej práci Martina Jurčíka využitím MLS funkcionality SELinux-u. Implementácia by mala pozostávať zo SELinux politiky a potrebných systémových nástrojov pre administrátora a/ alebo používateľov.

Vedúci: RNDr. Jaroslav Janáček, PhD.
Katedra: FMFI.KI - Katedra informatiky

Dátum zadania: 19.10.2010

Dátum schválenia: 19.10.2010 prof. RNDr. Branislav Rovan, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Bc. Martin Jurčík
Study programme:	Computer Science (Single degree study, master II. deg., full time form)
Field of Study:	9.2.1. Computer Science, Informatics
Type of Thesis:	Diploma Thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Using SELinux to Enforce Two-Dimensional Labelled Security Model with Partially Trusted Subjects

Aim: The goal of the thesis is to implement our two-dimensional labelled security model with partially trusted subjects in Linux operating system using SELinux mechanism. The author of the thesis is expected to improve the implementation presented in Martin Jurčík's bachelor thesis by utilizing multi-level security features of SELinux. The implementation is expected to consist of the SELinux policy definition and supporting tools for the system administrator and/or users.

Supervisor: RNDr. Jaroslav Janáček, PhD.
Department: FMFI.KI - Department of Computer Science

Assigned: 19.10.2010

Approved: 19.10.2010 prof. RNDr. Branislav Rovan, PhD.
Guarantor of Study Programme

.....
Student

.....
Supervisor

Declaration of Authorship

I hereby declare that this thesis represents my own work and effort. Where other sources of information have been used, they have been acknowledged.

Bratislava, 2012 Signature: . . . . . .

Acknowledgements

I would like to thank my supervisor RNDr. Jaroslav Janáček, PhD. for his guidance, support, and encouragement throughout writing this thesis.

Special thanks belong to my family for all their support.

Abstrakt

Autor:	Bc. Martin Jurčík
Názov diplomovej práce:	Using SELinux to Enforce Two-Dimensional Labelled Security Model with Partially Trusted Subjects
Škola:	Univerzita Komenského v Bratislave
Fakulta:	Fakulta matematiky, fyziky a informatiky
Katedra:	Katedra informatiky
Vedúci diplomovej práce:	RNDr. Jaroslav Janáček, PhD.
Rozsah práce:	90 strán

Bratislava, 2012

Cieľom tejto práce bolo implementovať náš bezpečnostný model s názvom Two-dimensional labelled security model with partially trusted subjects použitím SELinux mechanizmu v operačnom systéme Linux. Očakávali sme zlepšenie implementácie prezentovanej v bakalárskej práci využitím viacúrovňových bezpečnostných prvkov SELinuxu. Implementácia mala pozostávať zo SELinux politiky a podporných nástrojov pre administrátora systému a/alebo užívateľov. Podarilo sa nám nájsť takú implementáciu tohto modelu, ktorú je možné ľahko kombinovať s ľubovolnou už existujúcou SELinux politikou bez potreby špeciálnych podporných nástrojov.

Kľúčové slová: politika toku informácií, bezpečnostný model, SELinux politika, MLS SELinux, MCS SELinux

Abstract

Author:	Bc. Martin Jurčík
Title:	Using SELinux to Enforce Two-Dimensional Labelled Security Model with Partially Trusted Subjects
University:	Comenius University in Bratislava
Faculty:	Faculty of Mathematics, Physics and Informatics
Department:	Department of Computer Science
Supervisor:	RNDr. Jaroslav Janáček, PhD.
Number of pages:	90
Bratislava, 2012	

The goal of the thesis was to implement our two-dimensional labelled security model with partially trusted subjects in Linux operating system using SELinux mechanism. We expected to improve the implementation presented in bachelor thesis by utilizing multi-level security features of SELinux. The implementation was expected to consist of the SELinux policy definition and supporting tools for the system administrator and/or users. We found an implementation of this model that can be easily combined with an arbitrary SELinux policy without any special supporting tools.

Keywords: information flow policy, security model, SELinux policy, MLS SELinux, MCS SELinux

Contents

Introduction

Information security means protecting information and information systems from unauthorized access, use, disclosure, disruption, modification, perusal, inspection, recording or destruction.

The terms information security, computer security and information assurance are frequently used interchangeably. These fields are interrelated often and share the common goals of protecting the confidentiality, integrity and availability of information; however, there are some subtle differences between them.

These differences lie primarily in the approach to the subject, the methodologies used, and the areas of concentration. Information security is concerned with the confidentiality, integrity and availability of data regardless of the form the data may take: electronic, print, or other forms. Computer security can focus on ensuring the availability and correct operation of a computer system without concern for the information stored or processed by the computer. Information assurance focuses on the reasons for assurance that information is protected, and is thus reasoning about information security.

The goal of this thesis is to implement our two-dimensional labelled security model with partially trusted subjects in Linux operating system using SELinux mechanism. We are expected to improve the implementation presented in bachelor thesis by utilizing multi-level security features of SELinux. The implementation is expected to consist of the SELinux policy definition and supporting tools for the system administrator and/or users.

Chapter 1

Preliminaries

This chapter gives small overviews of framework that we use, and security model that we will implement. Section 1.1 offers a brief summary of Security-Enhanced Linux, which is our framework to implement our security model. We describe an overview about it, its evolution in time, and security benefits. Section 1.2 gives small overview of SELinux Reference Policy project. Section 1.3 onwards, Two-Dimensional labelled security model with partially trusted subjects will be brought to life.

1.1 Security-Enhanced Linux

Security-Enhanced Linux (SELinux) is a Linux feature that provides a mechanism for supporting access control security policies. SELinux also includes United States Department of Defense-style mandatory access controls, through the use of Linux Security Modules (LSM) in the Linux kernel. Its architecture strives to separate enforcement of security decisions from the security policy itself and streamlines the volume of software charged with security policy enforcement.

Security-Enhanced Linux implements the Flux Advanced Security Kernel (FLASK) integrated in some versions of the Linux kernel with a number of utilities designed to demonstrate the value of mandatory access controls to the Linux community and how such controls could be added to Linux. Such a kernel contains architectural components prototyped in the Fluke operating system. These provide general support for enforcing many kinds of mandatory access control policies, including those based on the concepts of Type Enforcement, Role-Based Access Control, and multilevel security. FLASK, in turn,

was based on DTOS, a Mach-derived Distributed Trusted Operating System, as well as Trusted Mach, a research project from Trusted Information Systems that had an influence on the design and implementation of DTOS.

SELinux was originally a development project from the National Security Agency (NSA) and others. As we mentioned above, the Flask architecture implements mandatory access control, which focuses on providing an administratively-defined security policy that can control all subjects and objects, basing decisions on all security-relevant information. In addition, Flask focuses on the concept of least privilege, which gives a process exactly the rights it needs to perform its given task.

The Flask model allows us to express a security policy in a naturally flowing manner, so that parts of the security rules are like parts in a sentence. In Flask, changes are supported so we can tune our policy. Added to this architecture in the security server are Type Enforcement (TE) and Role-Based Access Control (RBAC) security models, providing fine-grained controls that can be transparent to users and applications.

As a next step in the evolution of SELinux, the NSA integrated SELinux into the Linux kernel using the Linux Security Modules (LSM) framework. SELinux motivated the creation of LSM, at the suggestion of Linus Torvalds, who wanted a modular approach to security instead of accepting just SELinux into the kernel. Originally, the SELinux implementation used persistent security IDs (PSIDs) stored in an unused field of the ext2 inode. These numerical representations (i.e., non-human-readable) were mapped by SELinux to a security context label. Unfortunately, this required modifying each file system type to support PSIDs, so was not a scalable solution or one that would be supported upstream in the Linux kernel.

The next evolution of SELinux was as a loadable kernel module for the 2.4.<x> series of Linux kernels. This module stored PSIDs in a normal file, and SELinux was able to support more file systems. This solution was not optimal for performance, and was inconsistent across platforms.

Finally, the SELinux code was integrated upstream to the 2.6.x kernel, which has full support for LSM and has extended attributes (xattrs) in the ext3 file system. SELinux was moved to using xattrs to store security context information. The xattr namespace provides useful separation for multiple security modules existing on the same system.

1.1.1 SELinux Basics

SELinux implements two information flow mechanisms – Domain and Type Enforcement (DTE), and Multi-Level and Multi-Category Security (MLS and MCS). Every SELinux controlled object and every subject (process) is assigned a security label consisting of four parts – a user, a role, a type, and security level.

The SELinux user identities are not to be confused with the standard Linux user identities – they are independent attributes. They may be mapped in a 1:1 manner, but they do not have to. The SELinux roles are assigned to subjects, and the SELinux policy specifies the subject types (domains) that a role is authorized for, i.e. the set of types a subject with a given role may be assigned. The policy also specifies the set of roles a SELinux user is authorized to assume. We will not use specific roles for our model's policy, so we will assume that every user is authorized for a role that is authorized for all domains unless another security policy is in force, in which case the set of domains and roles is determined by the other policy.

The types are declared in the SELinux policy configuration. A type may be labelled by several attributes. An attribute name may be used in the policy configuration to represent the set of types that are labelled with the attribute. The access control rules specify permissions for triplets – the type of a subject, the type of an object and the class of the object. Classes are used to distinguish different sorts of objects, such as files, directories, processes . . . .

Another important concept of SELinux policy configuration is the concept of constraints. A constraint is specified for a set of classes and a set of permissions, and it consists of a boolean expression. If the expression evaluates to false, no operation from the set may be performed on an object of the specified class. The expression may compare SELinux user identities, roles and types of the subject and the object with each other, or with a set of values. The set of types may also be specified using the type attributes as stated above. The MLS component of SELinux adds another sort of constraints – MLS constraints. These constraints allow for comparison of security levels of subjects and objects. Multi-Category Security (MCS), in comparison to Multi-Level Security (MLS), ignores sensitivity levels, i.e. MCS always runs at a single level. SELinux MLS and SELinux MCS will be discussed later, when their time has come.

1.1.2 Summary

The security of a Linux system without SELinux depends on the correctness of the kernel, of all the privileged applications, and of each of their configurations. A problem in any one of these areas may allow the compromise of the entire system. In contrast, the security of a system kernel based on a SELinux depends primarily on the correctness of the kernel and its security-policy configuration. While problems with the correctness or configuration of applications may allow the limited compromise of individual user programs and system daemons, they do not pose a threat to the security of other user programs and system daemons or to the security of the system as a whole.

A Linux kernel integrating SELinux enforces mandatory access-control policies that confine user programs and system servers to the minimum amount of privilege they require to do their jobs. This reduces or eliminates the ability of these programs and daemons to cause harm when compromised (via buffer overflows or misconfigurations, for example). This confinement mechanism operates independently of the traditional Linux access control mechanisms. It has no concept of a "root" super-user, and does not share the well-known shortcomings of the traditional Linux security mechanisms.

Thus, SELinux provides a hybrid of concepts and capabilities drawn from mandatory access controls (MAC), mandatory integrity control (MIC), role-based access control (RBAC), and type enforcement (TE) architecture. Mandatory access controls allow an administrator of a system to define how applications and users can access different resources such as files, devices, networks and inter-process communication. With SELinux an administrator can differentiate a user from the applications a user runs. For example, the user shell or GUI may have access to do anything he wants with his home directory but if he runs a mail client the client may not be able to access different parts of the home directory, such as his ssh keys. The way that an administrator sets these permissions is with the centralized SELinux policy. The policy for SELinux is a binary representation that can be loaded into the kernel. It tells the system how different components on the system can interact and use resources. The policy typically comes from distribution that is used, but it can be updated on the end system to reflect different configurations or application behaviour.

- SELinux can potentially control, which activities are allowed for each user, process and daemon, with very precise specifications. However, it is mostly used to confine daemons like database engines or web servers that have more clearly defined data access and activity rights. A confined daemon that becomes compromised is thus

limited in the harm it can do. Ordinary user processes often run in the unconfined domain, not restricted by SELinux but still restricted by the classic Linux access rights.

- SELinux is a security enhancement to Linux that allows users and administrators more control over which users and applications can access which resources, such as files. Standard Linux access controls, such as file modes (-rwxr-xr-x) are modifiable by the user and applications that the user runs whereas SELinux access controls are determined by a policy loaded on the system and not changeable by careless users or misbehaving applications.
- SELinux also adds finer granularity to access controls. Instead of only being able to specify who can read, write or execute a file, for example, SELinux lets us specify who can unlink, append only, and move a file and so on. SELinux allows us to specify access to many resources other than files as well, such as network resources and interprocess communication (IPC).
- SELinux controls access between applications and resources. By using a mandatory security policy SELinux enforces the security goals of the system regardless of whether applications misbehave or users act carelessly. SELinux is capable of enforcing a wide range of security goals, from simply sandboxing applications to locking down network facing daemons and restricting users to only the resources they need to work.

1.2 SELinux Reference Policy

This section covers some background on SELinux Reference Policy project.

1.2.1 SELinux Reference Policy Project Overview

The SELinux Reference Policy project (refpolicy) is a complete SELinux policy that can be used as the system policy for a variety of systems and used as the basis for creating other policies. Reference Policy was originally based on the NSA example policy, but aims to accomplish many additional goals.

Reference Policy is under active development, with support and full time development staff from Tresys Technology. The current release is available on their web page [26] (section DownloadRelease).

1.2.2 SELinux Reference Policy Project Goals

Security is the reason for existence for SELinux policies and must, therefore, always be the first priority. The common view of security as a binary state (secure or not secure) is not a sufficient goal for developing a SELinux policy. In reality, different systems have different requirements and purposes and corresponding differences in the meaning of secure. What is a fundamental security flaw on one system might be acceptable, or even the primary functionality, of another. The challenge for a system policy is to support as many of these differing security goals as is practical. To accomplish this Reference Policy will provide:

- Strong Modularity: Central to the design of the policy is strict modularity. Accesses to resources are abstracted, and implementation details are encapsulated in the module.
- Security Goals: Clearly stated security goals will for each component of the policy. This will allow policy developers to determine if a given component meets their security needs.
- Documentation: The difficulty and complexity of creating SELinux policies has become the number one barrier to the adoption of SELinux. It also potentially reduces the security of the policies: a policy that is too complex to easily understand is difficult to make secure. Reference Policy will make aggressive improvements in this area by including documentation for modules and their interfaces as a critical part of the infrastructure.
- Development Tool Support: In addition to documentation, Reference Policy aims to make improvements in this area, making policies easier to develop, understand, analyze, and verify by adding interface call back traces, which can be used for debugging and graphical development tools.
- Forward Looking: Reference Policy aims to support a variety of policy configurations and formats, including standard source policies, MLS policies, and loadable policy modules all from the same source tree. This is done through the addition of infrastructure for automatically handling the differences between source and loadable module based policies and the additional MLS fields to all policy statements that include contexts.

- Configurability: Configuration tools that allow the policy developer to make important security decisions including defining roles, configuring networking, and trading legacy compatibility for increased security.
- Flexible Base Policy: A base policy that protects the basic operating system and serves as a foundation to the rest of the policy. This base policy should be able to support a variety of application policies with differing security goals.
- Application Policy Variations: Application policy variations that make different security tradeoffs. For example, two Apache policies might be created, one that is for serving read-only static content that is severely restricted, and another that is appropriate for dynamic content.
- Multi-Level Security: MLS is supported out-of-the-box without requiring destructive changes to the policy. It is possible to compile and MLS and non-MLS policy from the same policy files by switching a configuration option.

1.2.3 Modules

Modules are the principal organizing component in refpolicy. A module contains the policy for an application or related group of applications, private and shared resources, labelling information, and interfaces that allow other modules access to the module's resources. The majority of the global policy has been eliminated in refpolicy. Certain policy components, like users and object classes, are still global in refpolicy, but almost all TE policy is now contained within a module.

1.3 Two-Dimensional Labelled Security Model with Partially Trusted Subjects

In this section, we present our security model. The security model is based on PhD thesis [9] written by RNDr. Jaroslav Janáček, PhD..

The model contains two types of entities – subjects and objects. Objects are passive entities of the model – they represent information sources and destinations. Some typical examples of objects in an operating system are files, directories, communication objects (such as pipes, sockets . . .). Subjects are active entities of the model – they perform operations on objects. Typical subjects of an operating system are processes.

We will suppose these operations that a subject may perform on an object: read (and get attribute), write (and set attribute), create, and delete (may signal). The read and get attribute operations allow the subject to receive information from the object, and the write and set attribute operations allow the subject to send information to the object. Each object O and each subject S has several security attributes associated with it, and the model's information flow policy specifies whether a subject S is allowed to perform a given operation on an object O based on the security attributes of the subject and the security attributes of the object.

Our data classification scheme assumes that each object is assigned a confidentiality level and an integrity level. Each object is also assigned an identifier of a user that is the object's owner. Our model assumes there are at least three confidentiality and three integrity levels. As far as the confidentiality is concerned, we classify the data into three basic categories: public data (level 0), normal data – C-normal (level 1), and data that are sensitive regarding their confidentiality – Csensitive (level 2). The C-sensitive data are the data that their owner (a user) wishes to remain unreadable to the others regardless of the software the user uses, and even if the users makes some mistakes (such as setting wrong access rights for discretionary access control). As far as the integrity (or trustworthiness) of data is concerned, we also classify the data into three basic categories: potentially malicious data (level 0), normal data – I-normal (level 1), data that are sensitive regarding their integrity – I-sensitive (level 2). The requirements of the integrity protection of data are tightly coupled to the trustworthiness of the data. The trustworthiness of data can be thought of as a metric of how reliable the data are. If some data can be modified by anyone, they cannot be trusted not to contain wrong or malicious information. If some data are to be relied on, their integrity has to be protected. The potentially malicious data require no integrity protection, and can neither be trusted to contain valid information, nor can be trusted not to contain malicious content. The I-sensitive data are the data that their owner wishes to remain unmodified by the others regardless of the software the user uses, and even if the users makes some mistakes. The I-sensitive data are to be modifiable only under special conditions upon their owner's request. A special category of I-sensitive data is the category of the shared system files such as the programs, the libraries, various system-wide configuration files, the user database . . . .

Some of these files may be modifiable by the designated system administrator; some of them should be even more restricted. The basic idea of our model is to prevent unintended information flow from an object with a higher confidentiality/lower integrity level to an object with a lower confidentiality/higher integrity level.

Classical multi-level security models, such as Bell-La Padula or Biba, distinguish between untrusted and trusted subjects. Trusted subjects are allowed to violate the basic idea stated above. It turns out that in a typical small office or home desktop operating system too many subjects would have to be considered trusted in order to achieve an acceptable behaviour.

To overcome this problem, we divide subjects into three categories:

- untrusted subjects,
- partially trusted subjects, and
- trusted subjects.

A trusted subject is a subject that is trusted to enforce the information flow policy with intended exceptions by itself. An untrusted subject is a subject that is not trusted to enforce the information flow policy. It is assumed to perform any operations on any objects unless it is prevented from doing so by the operating system.

A partially trusted subject is

- trusted not to transfer information from a defined set of objects (designated inputs) at a higher confidentiality level to a defined set of objects (designated outputs) at a lower confidentiality level in a way other than the intended one, and
- trusted not to transfer information from a defined set of objects (designated inputs) at a lower integrity level to a defined set of objects (designated outputs) at a higher integrity level in a way other than the intended one, but
- not trusted not to transfer information between any other objects.

The sets of designated inputs and outputs regarding confidentiality are distinct from the sets regarding integrity. Any of the sets may be empty. A partially trusted subject, like a trusted one, can be used to implement an exception to the basic policy, because it can violate the policy (and it is trusted to do it only in an intended way). The most important difference between trusted and partially trusted subjects is in the level of trust. While trusted subjects are completely trusted to behave correctly, partially trusted subjects are only trusted not to abuse the possibility of the information flow violating the policy between a defined set of input objects and a defined set of output objects.

1.3.1 Formal Definition of the Information Flow Policy

Let $C = \{0, 1, \ldots, c_{max}\}$ be the set of confidentiality levels, $I = \{0, 1, \ldots, i_{max}\}$ be the set of integrity levels, L be the finite set of possible labels for objects, $0 \in L$ being the default label used for objects without an explicitly assigned label, and U be the final set of user identifiers. Let C and I be ordered so that 0 is the least sensitive level and c_{max} and i_{max} are the most sensitive levels.

Let us assume that each object O has the following attributes:

- $C_O \in C$ – the confidentiality level of the object,
- $I_O \in I$ – the integrity level of the object,
- $L_O \in L$ – the label of the object (used to define the input and output sets of objects for partially trusted subjects),
- $U_O \in U$ – the user identifier of the owner of the object.

Let us assume that each subject S has the following attributes:

- $CR_S \in C$ – the highest confidentiality level the subject can normally read from,
- $CW_S \in C$ – the lowest confidentiality level the subject can normally write to,
- $CRL_S \in C$ – the highest confidentiality level of a specially labelled object that the subject can read from,
- $CWL_S \in C$ – the lowest confidentiality level of a specially labelled object that the subject can write to,
- $CRLS_S \subseteq L$ – the set of labels of the objects that the subject can read from as a partially trusted subject,
- $CWLS_S \subseteq L$ – the set of labels of the objects that the subject can write to as a partially trusted subject,
- $IR_S \in I$ – the lowest integrity level the subject can normally read from,
- $IW_S \in I$ – the highest integrity level the subject can normally write to,
- $IRL_S \in I$ – the lowest integrity level of a specially labelled object that the subject can read from,

- $IWL_S \in I$ – the highest integrity level of a specially labelled object that the subject can write to,
- $IRLS_S \subseteq L$ – the set of labels of the objects that the subject can read from as a partially trusted subject,
- $IWLS_S \subseteq L$ – the set of labels of the objects that the subject can write to as a partially trusted subject,
- $CN_S \in C$ – the default confidentiality level of the objects created by the subject,
- $IN_S \in I$ – the default integrity level of the objects created by the subject,
- $LN_S \in L$ – the label of the objects created by the subject,
- $U_S \in U$ – the user identifier of the owner of the subject,
- $IRUS_S \subseteq U$ – the set of additional user identifiers of the users who are trusted by S to maintain trustworthy integrity levels on the objects they own (e.g. a special user designated to own the shared system libraries and programs).
- $CWUS_S \subseteq U$ – the set of additional user identifiers of the users who are trusted by S to maintain trustworthy confidentiality levels on the objects they own.

Let $C_{appr}, C_{shareable}, I_{shareable}$ be system-wide constants with the following meaning:

- $C_{appr} \in C$ be the highest confidentiality level for which the user may interactively approve a request to read from an object O by a subject S when $C_{appr} \geq C_O > CR_S$,
- $C_{shareable} \in C$ be the highest confidentiality level of an object that may be accessed by a subject with a different owner than the owner of the object, and
- $I_{shareable} \in I$ be the highest integrity level of an object that that may be modified by a subject with a different owner than the owner of the object.

Let us define the information flow policy protecting confidentiality and integrity of data as follows:

1. A subject S may **read** from an object O if $\mathbf{read}(S,O)$ is true, where

$$\begin{aligned}\mathbf{read}(S,O) \overset{\text{def}}{\iff} &[CR_S \geq C_O \vee (CRL_S \geq C_O \wedge L_O \in CRLS_S) \\ &\vee (C_{appr} \geq C_O \wedge \mathbf{UserApprovedRead}(S,O))] \qquad (1.1a)\\ \wedge &[IR_S \leq I_O \vee (IRL_S \leq I_O \wedge L_O \in IRLS_S)] \qquad (1.1b)\\ \wedge &[U_S = U_O \vee C_O \leq C_{shareable}] \qquad (1.1c)\\ \wedge &[U_S = U_O \vee U_O \in IRUS_S \vee IR_S \leq I_{shareable}] \qquad (1.1d)\end{aligned}$$

   where $\mathbf{UserApprovedRead}(S,O)$ is **true** if and only if the user (the owner of S) has approved the particular request to read from the object O by the subject S.

2. A subject S may **write** to an object O if $\mathbf{write}(S,O)$ is true, where

$$\begin{aligned}\mathbf{write}(S,O) \overset{\text{def}}{\iff} &[CW_S \leq C_O \vee (CWL_S \leq C_O \wedge L_O \in CWLS_S)] \qquad (1.2a)\\ \wedge &[IW_S \geq I_O \vee (IWL_S \geq I_O \wedge L_O \in IWLS_S)] \qquad (1.2b)\\ \wedge &[U_S = U_O \vee I_O \leq I_{shareable}] \qquad (1.2c)\\ \wedge &[U_S = U_O \vee U_O \in CWUS_S \vee CW_S \leq C_{shareable})] \qquad (1.2d)\end{aligned}$$

3. A subject S may **create** a new object O within (or related to) an object P if $\mathbf{create}(S,P)$ is true, where

$$\begin{aligned}\mathbf{create}(S,P) \overset{\text{def}}{\iff} &\mathbf{read}(S,P) \qquad (1.3a)\\ \wedge &\mathbf{write}(S,P) \qquad (1.3b)\end{aligned}$$

The attributes of the new object will be set as follows:

$$C_O := \begin{cases} CWL_S & \text{if } L_P \in CWLS_S \\ CN_S & \text{otherwise} \end{cases} \tag{1.3c}$$

$$I_O := \begin{cases} IWL_S & \text{if } L_P \in IWLS_S \\ IN_S & \text{otherwise} \end{cases} \tag{1.3d}$$

$$L_O := LN_S \tag{1.3e}$$

$$U_O := U_S \tag{1.3f}$$

4. A subject S may **delete** an object O from (or related to) an object P if $\mathbf{delete}(S,O,P)$ is true, where

$$\mathbf{delete}(S,O,P) \overset{\text{def}}{\iff} \mathbf{read}(S,P) \tag{1.4a}$$

$$\wedge \mathbf{write}(S,P) \tag{1.4b}$$

$$\wedge \mathbf{write}(S,O) \tag{1.4c}$$

5. Each untrusted subject S must satisfy:

$$CW_S = CWL_S \geq CR_S = CRL_S \tag{1.5a}$$

$$IW_S = IWL_S \leq IR_S = IRL_S \tag{1.5b}$$

$$CWLS_S = CRLS_S = IWLS_S = IRLS_S = \emptyset \tag{1.5c}$$

$$CN_S \geq CW_S \tag{1.5d}$$

$$IN_S \leq IW_S \tag{1.5e}$$

$$LN_S = 0 \tag{1.5f}$$

6. Each partially trusted subject S must satisfy:

$$CW_S \geq CR_S \tag{1.6a}$$
$$CW_S \geq CRL_S \tag{1.6b}$$
$$CWL_S \geq CR_S \tag{1.6c}$$
$$IW_S \leq IR_S \tag{1.6d}$$
$$IW_S \leq IRL_S \tag{1.6e}$$
$$IWL_S \leq IR_S \tag{1.6f}$$
$$CN_S \geq CW_S \tag{1.6g}$$
$$IN_S \leq IW_S \tag{1.6h}$$

The above rules fulfil the policy objectives on the condition that:

$$\begin{aligned} C &= \{0,1,2\} \\ I &= \{0,1,2\} \\ C_{appr} &= 1 \\ C_{shareable} &= 1 \\ I_{shareable} &= 1 \end{aligned}$$

with the meaning of the confidentiality levels:

$$\begin{aligned} &0 - \text{public}, \\ &1 - \text{C-normal}, \\ &2 - \text{C-sensitive}, \end{aligned}$$

and the meaning of the integrity levels:

$$\begin{aligned} &0 - \text{potentially malicious}, \\ &1 - \text{I-normal}, \\ &2 - \text{I-sensitive}. \end{aligned}$$

Additional Operations

- A subject S may set the confidentiality level of an object O to c, and the integrity level of O to i if **reclassify**(S,O,c,i) is true, where

$$\begin{aligned}
\textbf{reclassify}(S,O,c,i) \overset{\text{def}}{\iff} & [C_O \leq CR_S \wedge C_O \geq CW_S \wedge c \geq CW_S] && \text{(1.7a)}\\
& \wedge [I_O \geq IR_S \wedge I_O \leq IW_S \wedge i \leq IW_S] && \text{(1.7b)}\\
& \wedge \textbf{CanRevoke}(O) && \text{(1.7c)}\\
& \wedge U_O = U_S && \text{(1.7d)}\\
& \wedge L_O = LN_S && \text{(1.7e)}
\end{aligned}$$

- A subject D (the debugger) may use the debugging interface to debug a subject S if **debug**(D,S) is true, where

$$\begin{aligned}
\textbf{debug}(D,S) \overset{\text{def}}{\iff} & CR_D \geq \max\{CR_S, CW_S\} && \text{(1.8a)}\\
& \wedge CW_D \leq \min\{CR_S, CW_S\} && \text{(1.8b)}\\
& \wedge IR_D \leq \min\{IR_S, IW_S\} && \text{(1.8c)}\\
& \wedge IW_D \geq \max\{IR_S, IW_S\} && \text{(1.8d)}\\
& \wedge U_D = U_S && \text{(1.8e)}
\end{aligned}$$

- A subject S may send a signal to a subject R if **maysignal**(S,R) is true, where

$$\begin{aligned}
\textbf{maysignal}(S,R) \overset{\text{def}}{\iff} & CW_S \leq CR_R && \text{(1.9a)}\\
& \wedge IW_S \geq IW_R && \text{(1.9b)}\\
& \wedge U_S = U_R && \text{(1.9c)}
\end{aligned}$$

Changing the subject's security attributes

No subject may be able to modify its attributes in a way that allows it to perform more operations. The following rules satisfy the requirement:

1. A subject S may change CN_S to c if $\mathbf{setCN}(S,c)$ is true, where

$$\mathbf{setCN}(S,c) \overset{\text{def}}{\iff} c \geq CW_S \tag{1.10}$$

2. A subject S may change IN_S to i if $\mathbf{setIN}(S,i)$ is true, where

$$\mathbf{setIN}(S,i) \overset{\text{def}}{\iff} i \leq IW_S \tag{1.11}$$

3. A subject S may change CR_S to c if $\mathbf{setCR}(S,c)$ is true, where

$$\mathbf{setCR}(S,c) \overset{\text{def}}{\iff} c \leq CR_S \tag{1.12}$$

4. A subject S may change CW_S to c if $\mathbf{setCW}(S,c)$ is true, where

$$\mathbf{setCW}(S,c) \overset{\text{def}}{\iff} c \geq CW_S \tag{1.13}$$

5. A subject S may change IR_S to i if $\mathbf{setIR}(S,i)$ is true, where

$$\mathbf{setIR}(S,i) \overset{\text{def}}{\iff} i \geq IR_S \tag{1.14}$$

6. A subject S may change IW_S to i if $\mathbf{setIW}(S,i)$ is true, where

$$\mathbf{setIW}(S,i) \overset{\text{def}}{\iff} i \leq IW_S \tag{1.15}$$

7. A subject S may change CRL_S to c if $\mathbf{setCRL}(S,c)$ is true, where

$$\mathbf{setCRL}(S,c) \overset{\text{def}}{\iff} c \leq CR_S \tag{1.16}$$

8. A subject S may change CWL_S to c if $\mathbf{setCWL}(S,c)$ is true, where

$$\mathbf{setCWL}(S,c) \overset{\text{def}}{\iff} c \geq CW_S \tag{1.17}$$

9. A subject S may change IRL_S to i if **setIRL**(S,i) is true, where

$$\textbf{setIRL}(S,i) \overset{\text{def}}{\iff} i \geq IR_S \tag{1.18}$$

10. A subject S may change IWL_S to i if **setIWL**(S,i) is true, where

$$\textbf{setIWL}(S,i) \overset{\text{def}}{\iff} i \leq IW_S \tag{1.19}$$

11. When a subject S creates a new subject S', the security attributes of S' must be equal to those of S.

Chapter 2

Means of Implementation

In this chapter, we present a variety of means that will help us with efficient implementation. The section 2.1 gives small overview of SELinux Multi-Level Security. In section 2.2, we give overview of MLS constraints that allow us to restrict specified permissions for specified object classes by defining constraints. Section 2.3 gives small overview of SELinux Multi-Category Security. Section 2.4 gives small overview of one of Type Enforcement rules.

2.1 SELinux MLS

In the following section, we will look at SELinux Multi-Level Security.

Type Enforcement is far and away the most important mandatory access control (MAC) mechanism that SELinux introduces. However, in some situations, primarily for a subset of classified government applications, traditional multi-level security (MLS) MAC coupled with Type Enforcement is valuable. In recognition of these situations, SELinux has always had some form of MLS capability included. The MLS features are optional and generally the less important of the two MAC mechanisms in SELinux. For the vast majority of security applications, including many if not most classified data applications, Type Enforcement is the best-suited mechanism for enhanced security.

2.1.1 History of Multi-Level Security

The MLS functionality in SELinux is being developed as part of the Common Criteria LSPP certification work. The LSPP work aims to get LSPP , RBAC , and CAPP certification at EAL 4+. James Morris has a lot of background information on the LSPP work on his blog [19].

Multi-Level Security (MLS) policy was first formalized by Bell and LaPadula (BLP) [3] in the 70's of the last century. Systems implemented with MLS policies were primarily used to enforce confidentiality. In the 80's and 90's of the last century, the Compartmented Mode Workstation used MLS as the primary Mandatory Access Control (MAC) mechanism for evaluation to the Orange Book [5]. Today, MLS is still a key requirement to meet the Common Criteria [6] Label Security Protection Profile (LSPP) and future Medium Robustness Multi-Level Operating System Profiles for Common Criteria.

2.1.2 Security Context and MLS

The main visible component of SELinux is the security context. The security context is utilized by SELinux to work within the Flask architecture. The context is made up of four fields: SELinux User, Role, Type, and MLS Range. The last portion, MLS Range, is an optional component.

The *security level* used by MLS systems is a combination of a hierarchical *sensitivity* and a set (including the null set) of nonhierarchical *categories*. These sensitivities and categories are used to reflect real information confidentiality or user clearances. In most SELinux policies, the sensitivities (s0, s1, ...) and categories (c0, c1, ...) are given generic names, leaving it to userspace programs and libraries to assign user-meaningful names. For example, s0 might be associated with UNCLASSIFIED and s1 with SECRET.

To support MLS, the security context is extended to include security levels as such these:

```
user:role:type:sensitivity[:categories][-sensitivity[:categories]]
```

Notice that the MLS security context must have at least one security level (which is composed of a single sensitivity and zero or more categories), but can include two security levels. These two security levels are called *low* (or current for processes) and *high* (or clearance for processes), respectively. If the high security level is missing, it is considered to be the same value as the low (the most common situation). In practice, the low and high

security levels are usually the same for most objects and processes. A range of levels is typically used for processes that are considered trusted subjects (that is, a process trusted with the ability to downgrade information) or multilevel objects such as directories that might contain objects of differing security levels.

2.1.3 Security Context with MLS

The security context is extended with two additional fields: a low and high security level. A security level itself has two fields: a sensitivity and a set of categories. Sensitivities are **strictly hierarchical** reflecting an ordered data sensitivity model, such as Top Secret, Secret, and Unclassified in government classification controls. Categories are **unordered**, reflecting the need for data compartmentalization. The basic idea is that we need both a high enough sensitivity clearance and the right categories to access data.

The security levels are not hierarchical but rather governed by a dominance relationship. Unlike strict ordering where a level is either higher than, equal to, or lower than another level, in a dominance relationship, there is a fourth state called incomparable (also known as noncomparable). What causes security levels to be related via dominance rather than equality are the categories, which have no hierarchical relationship to one another. As a result, the four dominance operators that can relate two MLS security levels are as follows:

- dom (dominates): SL1 dom SL2 if the sensitivity of SL1 is higher or equal to the sensitivity of SL2, and the categories of SL1 are a superset of the categories of SL2.
- domby (dominated by): SL1 domby SL2 if the sensitivity of SL1 is lower than or equal to the sensitivity of SL2, and the categories of SL1 are a subset of the categories of SL2.
- eq (equals): SL1 eq SL2 if the sensitivities of SL1 and SL2 are equal, and the categories of SL1 and SL2 are the same set.
- incomp (incomparable or noncomparable): SL1 incomp SL2 if the categories of SL1 and SL2 cannot be compared (that is, neither is a subset of the other).

Given the dominance relationship, a variation of the Bell-La Padula model is implemented in SELinux where a process can "read" an object if its current security level dominates the security level of the object, and "write" an object if its current security level is dominated

by the security level of the object (and therefore read and write the object only if the two security levels are equal).

The MLS constraints in SELinux are in addition to the TE rules. If MLS is enabled, both checks must pass (in addition to standard Linux access control) for access to be granted.

2.1.4 Defining Security Levels

In an SELinux policy, we define sensitivities using the sensitivity statement, as follows:

```
sensitivity    s0;
sensitivity    s1;
sensitivity    s2;
sensitivity    s3;
```

These statements define four sensitivities called s0, s1, s2, and s3. These names are a typical generic sensitivity naming convention in SELinux. We could use any name we want here. The sensitivity statement also supports the ability to associate additional alias names with a sensitivity that will be treated the same as the core sensitivity name.

For example:

```
sensitivity s0 alias unclassified;
```

Because sensitivities must be hierarchically related, we must specify in the policy the hierarchy of sensitivities using the dominance statement, as follows:

```
dominance  s0 s1 s2 s3          # s0 is "low" and s3 "high"
```

The dominance statement lists the sensitivity names in order from lower to highest. Thus, in our example, s0 is lower than s1, which is lower than s2, and so forth.

The ordering of sensitivities is from lower to highest. All defined sensitivities must be contained within the dominance statement in order to define the complete sensitivity hierarchy.

Categories are defined in a similar manner as sensitivities using the category statement. As with sensitivities, categories may also have alias names. Unlike sensitivities, categories are not hierarchically related (or related at all). So, there is no need to define any explicit relationship between categories. The following statements are examples of the category statement:

```
category    c0  alias   blue;
category    c1  alias   red;
category    c2  alias   green;
category    c3  alias   orange;
category    c4  alias   white;
```

The final step in defining security levels in the policy language is to define allowed security level combinations using the level statement. The level statement dictates how we may associate categories with sensitivities. Remember that a combination of a single sensitivity and a set of categories constitute a security level. Here are some examples of the level statement:

```
level  s0:c0.c4;
level  s1:c0.c4;
level  s2:c0.c4;
level  s3:c0.c4;
```

These statements enable us to combine any of the defined categories with all the defined sensitivities from our earlier examples. We would generally have a single level statement for each defined sensitivity that identifies the categories that may be associated with each sensitivity in a valid security level. In the preceding example, we associated all five defined categories (c0.c4) with all four defined sensitivities. We can be more restrictive in this association:

```
level  s0:c0.c2;
level  s1:c0.c2,c4;
```

In this example, s0 may be associated only with categories c0, c1, and c2; and s1 with categories c0, c1, c2 and c4 (but not c3). By now, we should have noticed that a dot (.) indicates an inclusive range of categories, and a comma (,) indicates a noncontiguous list of categories.

Note 1: Just because ranges of categories are specified using the range operator (.), this does not mean that categories are hierarchically related. Instead, the range operator is just a convenient way to refer to a set of categories. The ordering of the categories for the range operator is just the order in which they are declared and has nothing to do with any intrinsic ordering implied by their names.

So, for example, if we declare that categories in the order c1, c0, and c2, the expressions c0.c2 would mean c0 and c2, and **not** c1. The level statement defines what

combinations of sensitivities and categories constitute an acceptable security level for the MLS portion of the SELinux policy.
Note 2: For a security context to be **valid**, the *high* level **must always dominate** the *low* level. In addition, the categories associated with the sensitivities **must be valid** per the level statements in the policy.
So, for example, if we have the previous level statements:

```
level s0:c0.c2;
level s1:c0.c2,c4;
```

and user_u, user_r, and user_t are valid user, role, and type identifiers, the following security contexts are invalid:

```
user_u:user_r:user_t:s0-s0:c2,c4
user_u:user_r:user_t:s0:c0-s0:c2
```

The first security context is invalid because category *c4* is invalid for sensitivity *s0*. Second security context is invalid because the *high* security level does not dominate the *low* security level.

2.1.5 Range_transition Statement

The range_transition statement is primarily used by the init process or administration commands to ensure processes run with their correct MLS range (for example init would run at *s15:c0.c255* and needs to initialise / run other processes at their correct MLS range). The statement was enhanced in Policy version 21 to accept other object classes.
The statement definition is

```
range_transition source_domain target_exectype : class new_mls_range;
```

Where:

- range_transition - The range_transition keyword.
- source_domain - A source process domain (as only the process object class is supported).
- target_exectype - A target executable type or attribute. (i.e. an identifier for a file that has the execute permission set.

- class - The optional object class keyword (this allows policy versions 21 and greater to specify a class other than the default of process).
- new_mls_range - The new MLS range for the object class.

Example:

```
# A range_transition statement from the MLS Reference Policy
# showing that a process anaconda_t can transition between
# systemLow (s0) and systemHigh (s15:c0.c255) depending
# on calling applications level.

range_transition anaconda_t init_script_file_type:
process s0 - s15:c0.c255;

# Two range_transition statements from the MLS Reference Policy
# showing that init will transition the audit and cups daemon
# to s15:c0.c255 (that is the lowest level they can run at).

range_transition initrc_t auditd_exec_t:process s15:c0.c255;
range_transition initrc_t cupsd_exec_t:process s15:c0.c255;
```

2.2 MLS Constraints

SELinux supports two MLS constraint statements, mlsconstrain and mlsvalidatetrans, which together enable us to specify the optional MLS access enforcement rules. These two statements are identical to their non-MLS counterparts except that they allow us to also express constraints based on the security levels of a security context. We may only use the MLS constraints in policies that have the optional MLS features enabled.

2.2.1 Mlsconstrain Statement

The *mlsconstrain* statement is based on the *constrain* statement. Detailed information about the *constrain* statement can be found in our bachelor's thesis [13]. The *mlsconstrain* statement adds new keywords for stating constraints based on the low and high security levels of the source (l1 and h1) and target (l2 and h2).

The *mlsconstrain* statement allows us to restrict specified permissions for specified object classes by defining constraints based on relationships between source and target security contexts that include the optional MLS features (that is, high and low security levels). The full syntax for the mlsconstrain statement is as follows:

```
mlsconstrain class_set perm_set expression ;
```

Where:

- class_set - One or more object classes. Multiple object classes must be separated by spaces and enclosed in braces ({ }) for example, file lnk_file. The special operators ∗, ~, and - are not allowed in class sets for this statement.
- perm_set - One or more permissions. All permissions must be valid for all object classes in the class_set. Multiple permissions must be separated by spaces and enclosed in braces ({ })for example, read create. The special operators ∗, ~, and - are not allowed in class sets for this statement.
- expression - A Boolean expression of the constraint.

The Boolean expression syntax supports the following keywords:

- t1, r1, u1, l1, h1 Source type, role, user, low level, and high level, respectively
- t2, r2, u2, l2, h2 Target type, role, user, low level, and high level, respectively

Constraint expression syntax also supports the following operators:

- == Set member of or equivalent.
- != Set not member of or not equivalent.
- eq (Roles and security level keywords only) equivalent.
- dom (Roles and security level keywords only) dominates.
- domby (Role and security level keywords only) not dominated by.
- incomp (Role and security level keywords only) incomparable.

The complete semantic meaning and allowed parameters for each operator are described in [1].

2.2.2 Mlsvalidatetrans Statement

The *mlsvalidatetrans* statement is similar to the *validatetrans* statement except that it introduces the six keywords l1 and h1, l2 and h2, and l3 and h3, meaning **old** low and high security levels, **new** low and high security levels, and **the source process** low and high security levels, respectively. The other difference between the two statements is that the mlsvalidatetrans statement is more commonly used to support an MLS policy than the validatetrans statement is in a typical TE policy. The mlsvalidatetrans statement **restricts the ability to change** the security context of specified supported objects by defining constraints-based relationships with old and new security contexts and the security context of the source process. The full syntax for the mlsvalidatetrans statement is as follows:

```
mlsvalidatetrans class_set expression ;
```

Where:

- class_set - One or more object supported classes. Multiple object classes must be enclosed in braces ({ })for example, file lnk_file. Currently, only permanent filesystem object classes are supported.
- expression - A Boolean expression of the constraint.

The Boolean expression syntax supports the following keywords:

- t1, r1, u1, l1, h1 Old type, role, user, low level, and high level, respectively
- t2, r2, u2, l2, h2 New type, role, and user, low level, and high level, respectively
- t3, r3, u3, l3, h3 Process type, role, user, low level, and high level, respectively

Note: Validatetrans and mlsvalidatetrans constraint statements support only filesystem objects; specifically, dir, file, lnk_file, chr_file, blk_file, sock_file, and fifo_file object classes.

2.3 SELinux MCS

Multi-Category Security (MCS) is in fact an adaptation of Multi-Level Security (MLS). It re-uses much of the MLS framework in SELinux, including the MLS label field, MLS kernel code, MLS policy constructs, labelled printing and label encoding/translation.

There are a few major differences between MCS and MLS:

- MCS ignores sensitivity levels. Everything is labelled with the same sensitivity, *s0*, which makes the idea of sensitivity levels effectively disappear.
- MCS discards the Bell La-Padula (BLP) security model. BLP properties such as No-Write-Down, which is designed to prevent leakage from high security levels to low security levels, are often confusing and break many assumptions of existing software. Consider root not being able to write to /tmp, as one of many possible problems relating to shared data.
- MCS is discretionary, similar to standard Unix DAC logic. The current implementation provides users with MCS control over their own files. This should map more readily to more general cases, reflecting the typical discretionary nature of real-world security outside of the Mil/Gov etc. environment. It is possible to adjust MCS to make it less discretionary, but really, that is what MLS is for.
- MCS always runs at a single level. The current level of all subjects running on the system is *s0*. When a subject is granted access to categories (remember: a security level is a combination of sensitivity and categories), this is done by adding categories to the high range clearance of the subject. The high range clearance refers to the maximum security level that the subject can run at. Under MCS, the subject will never run at anything other than *s0*, so the high range clearance is merely used to determine which categories the subject has access to when performing an access control decision. This is similar to Unix supplementary groups: a process can have access to several supplementary groups but not be running in any of them. Similarly, under MCS, a process can have access to a set of categories, but not be running at a security level which includes them. Having everything running at the same level vastly simplifies things.

MCS uses MLS technology, but is not MLS. Both traditional DAC and TE rules are consulted before MLS or MCS rules.

2.4 TE Transition Rules

TE Transition rules are one of the Type Enforcement rules. The Type Enforcement rules define what access control privileges are allowed for processes.

A *transition decision*, also referred to as a *labelling decision*, determines which security context will be assigned for a requested operation. There are two main types of transition. Firstly, there is a transition of process domains which is used when we execute a process of a specified type. Secondly, there is a transition of file type used when we create a file under a particular directory.

TE transition rules specify the new domain for a process or the new type for an object. In either case, the new type is based on a pair of types and a class. For a process, the first type, referred to as the source type, is the current domain and the second type, referred to as the target type, is the type of the executable. For an object, the source type is the domain of the creating process and the target type is the type of a related object, e.g. the parent directory for files.

2.4.1 Type_transition Statement

The type_transition statement specifies the labelling and object creation allowed between the source_type and target_type when a Domain Transition is requested.
Example - Domain Transition:

```
type_transition initrc_t acct_exec_t:process acct_t
```

Example - Object Transition:

```
type_transition acct_t var_log_t:file wtmp_t;
```

Chapter 3

Previous Implementation

In previous chapter, we described the means of implementation generally. Later, we will use these means in our new solution. In this chapter, we present our previous solution of implementation of our model by using SELinux. Section 3.1 offers a brief summary of our previous work.

3.1 Bachelor's Thesis

In this section, we present our previous solution of implementation of our model by using SELinux, namely Domain and Type Enforcement (DTE). In our previous solution, we encoded our model's attributes for objects and subjects as SELinux type attributes, and we used constraints to enforce the rules of our policy.

The objective of our bachelor's thesis was to create a prototype SELinux policy. We created a prototype policy for SELinux and we demonstrated the basic principles of Two-Dimensional Labelled Security Model with Partially Trusted Subjects. There we showed that the using of SELinux to enforce this security model is feasible. We presented several applications examples.

Now we describe only the basic ideas of this solution. For each entity of the operating system, we defined unique type with corresponding attributes, except some aliases of course. These attributes corresponded with level of confidentiality and integrity.

A standard policy configuration consists of the following top-level components:

1. FLASK definitions
2. Type Enforcement (TE) and Role-Based Access Control (RBAC) declarations and rules
3. User declarations
4. Constraint declarations
5. Security context specification

3.1.1 Previous Implementation

In our previous implementation, we used DTE types to encode the security attributes of objects and subjects. We needed a new DTE type for each combination of the values of the attributes. It led to a rather large number of types, and consequently to long expressions in the rules. If we use only three confidentiality and three integrity levels and L labels, we need $3^2 \cdot L$ types for objects, and $3^4 = 81$ types for trusted and untrusted subjects. We used to declare a new type for each partially trusted subject, because a general solution based on all possible combinations of the attribute values would make the total number of types infeasible. All the types were tagged with type attributes that were used in the expressions for the operations (see fig. 3.1). If we do not take partially trusted subjects into account, the number of type attributes is linear with respect to the number of security levels, the number of types is polynomial (power of 4), and the number of substatements in the rules is linear.

We defined the constraints for basic operations, namely **read**(S,O) and **write**(S,O), and for remaining operations, namely **create**(S,O), **debug**(D,S), and **maysignal**(S,R). The operation *get object's attributes* **getattr**(S,O) is equivalent to operation **read**(S,O) and the operation *set object's attributes* **setattr**(S,O) is equivalent to operation **write**(S,O). The corresponding permissions in SELinux for **read**(S,O) and **getattr**(S,O) are *read* and *getattr*, for **write**(S,O) and **setattr**(S,O) are *write* and *setattr*, for **create**(S,O) are *create* and *link*, for **debug**(D,S) is *ptrace*, and for **maysignal**(S,R) is corresponding class *process* and its permissions *sigkill*, *sigstop* and *signal*. For all **changing the subject's security** attributes operations is corresponding permission class *process* and permissions *transition*.

```
attribute C0;
attribute C1;
attribute C2;
attribute I0;
attribute I1;
attribute I2;
type Obj_C0I0, C0, I0;
type Obj_C0I1, C0, I1;
...
attribute CR0;
attribute CR1;
...
type Subj_CR0CW0IR0IW0, CR0, CW0, IR0, IW0;
...
constrain object_classes read_perms (
   (T2 == C2 and T1 == CR2)
or (T2 == C1 and T1 == {CR1 CR2})
or (T2 == C0)
... [ rules for partially trusted subjects ]
);
constrain object_classes read_perms (
   (T2 == I0 and T1 == IR0)
or (T2 == I1 and T1 == {IR1 IR0})
or (T2 == I2)
... [ rules for partially trusted subjects ]
);
```

Figure 3.1: Previous SELinux constraints enforcing our policy

More detailed information about this solution of implementation can be found in our bachelor's thesis [13].

We used TE transition rules to implement *create* mechanism, i.e. when a subject creates a new object (file) under a particular directory, the new object should have an exactly defined type. The section 2.4 describes the TE transition rules.

3.1.2 Disadvantages

From our point of view, unfortunately, the previous implementation has a number of limitations or substantial disadvantages, and thus it makes the implementation feasible, but it does not apply in practice easily.

The most important disadvantages are these:

1. Long expressions in the rules
2. The number of substatements in the rules is linear
3. Unique name for type for each of possible combinations of the attribute values
4. TE Transition rules for each of possible combinations of the subjects and objects that are used in *create* mechanism

The **first** and **second** disadvantages from our list are showed in fig. 3.2. This figure contains a detailed description of rules for operation **read(S,O)**. These rules contain substatements for partially trusted subjects and only two labels. For better understanding, we show a brief version of these rules in fig. 3.3. It is easy to see that the number of substatements for partially trusted subjects depends on number of labels for objects. In this case, there are only these labels: L0 and L1, respectively.

For the **third** disadvantage, suppose the following case. We have a special type for certain subject, e.g. browser. This subject has a type *browser1_t* with corresponding attributes values. If we need that this subject has a type with a special type with another attributes values, we will define a new type, e.g. *browser2_t*. Each type must have its own unique name.

Finally, for the **fourth** disadvantage, suppose this case. As stated in previous section, we used TE transition rules to implement *create* mechanism, i.e. when a subject creates a new object (a file) under a particular directory, the new object should have an exactly defined type. We have a special type for certain subject, e.g. a text editor. This subject has a type *my_text_editor_t* with corresponding attributes values. We have some directories in our filesystems, too. If we need that this subject creates a new file with a proper security type in each of these directories, we will define a new TE transition rule for each combination of this subject and all types of directories.

Fig. 3.4 shows an example of these type transition rules, where my_text_editor_t is type for subject, dir1, . . . , dirN are types for directories, and sec_type1_t, . . . , sec_typeN_t are proper types for new file in these directories.

```
constrain { blk_file chr_file sock_file } { read } (
(T2 == C2 and T1 == CR2) or (T2 == C1 and T1 == {CR1 CR2})
 or (T2 == C0)
or
(T2 == L0 and T1 == CRLS0 and T2 == C2 and T1 == CRL2)
 or
(T2 == L0 and T1 == CRLS0 and T2 == C1 and T1 == {CRL1 CRL2})
 or
(T2 == L1 and T1 == CRLS1 and T2 == C2 and T1 == CRL2)
 or
(T2 == L1 and T1 == CRLS1 and T2 == C1 and T1 == {CRL1 CRL2})
);

constrain { blk_file chr_file sock_file } { read } (
(T2 == I0 and T1 == IR0) or (T2 == I1 and T1 == {IR1 IR0})
 or (T2 == I2)
or
(T2 == L0 and T1 == IRLS0 and T2 == I0 and T1 == IRL0)
 or
(T2 == L0 and T1 == IRLS0 and T2 == I1 and T1 == {IRL1 IRL0})
 or
(T2 == L1 and T1 == IRLS1 and T2 == I0 and T1 == IRL0)
 or
(T2 == L1 and T1 == IRLS1 and T2 == I1 and T1 == {IRL1 IRL0})

);

constrain { blk_file chr_file sock_file } { read } (
(U1 == U2 or T2 == {C0 C1})
and
(U1 == U2 or T2 == {I0 I1} or U2 == {root system_u})
);
```

Figure 3.2: Previous SELinux constraints enforcing the operation **read(S, O)** with the partially trusted subject's rules for only 2 labels

```
constrain { blk_file chr_file sock_file } { read } (
(T2 == C2 and T1 == CR2) or (T2 == C1 and T1 == {CR1 CR2})
 or (T2 == C0)
or
(
((T2 == L0 and T1 == CRLS0) or (T2 == L1 and T1 == CRLS1))
 and
((T2 == C2 and T1 == CRL2) or (T2 == C1 and T1 == {CRL1 CRL2}))
)
);

constrain { blk_file chr_file sock_file } { read } (
(T2 == I0 and T1 == IR0) or (T2 == I1 and T1 == {IR1 IR0})
 or (T2 == I2)
or
(
((T2 == L0 and T1 == IRLS0) or (T2 == L1 and T1 == IRLS1))
 and
(( T2 == I0 and T1 == IRL0) or (T2 == I1 and T1 == {IRL1 IRL0}))
)
);

constrain { blk_file chr_file sock_file } { read } (
(U1 == U2 or T2 == {C0 C1})
and
(U1 == U2 or T2 == {I0 I1} or U2 == {root system_u})
);
```

Figure 3.3: Compact version of SELinux constraints enforcing the operation **read(S,O)**

```
type_transition my_text_editor_t dir1_t:file sec_type1_t;
type_transition my_text_editor_t dir2_t:file sec_type2_t;
...
type_transition my_text_editor_t diri_t:file sec_typei_t;
...
type_transition my_text_editor_t dirN_t:file sec_typeN_t;
```

Figure 3.4: Example of type transition rules

Chapter 4

New Ideas and Solutions

In this chapter, we provide our new implementation of our security model. We also describe our ideas, which led us to goal of this thesis. We describe the evolution steps of our implementation.

4.1 Multi-Level or Multi-Category Security

As the first step, we analyzed MLS components of SELinux. We found two possible ways how to propose new solution and/or improve our previous solution, respectively. In both cases, we encode our model's attributes for objects and subjects as MLS SELinux attributes. We found these possibilities:

1. We use pure SELinux MLS (see section 2.1).
2. We use its special case that is SELinux MCS (see section 2.3).

MLS SELinux offers us better possibilities because we can use sensitivities and categories together. On the other hand, we can use only categories in MCS SELinux. Thus, we decided for MLS SELinux.

4.2 Multi-Level Security Way-1.0

In this section, we present on our first attempt of implementation that uses MLS components of SELinux. The ideas are based on utilizing MLS components of SELinux in addition to the DTE component. We presented this solution in [14].

4.2.1 Solution

Our new idea is based on using the security levels to encode our attributes of model, and on using the dominance relation described below to evaluate the rules of our policy.

A SELinux security context is typically represented as:

*system_u:system_r:kernel_t:**s0:c0.c255-s15:c0.c255***

The *security level* used by MLS systems is a combination of a hierarchical sensitivity and a set (including the empty set) of nonhierarchical categories. Notice that the MLS security context must have at least one security level(which is composed of a single sensitivity and zero or more categories), but can include two security levels. These two security levels are called *low* and *high*, respectively.

Sensitivities are defined in SELinux policy using the **sensitivity** statements. Sensitivities are strictly hierarchical and therefore we must specify the hierarchy of sensitivites using the **dominance** statement. Sensitivities can be compared using the relational operators $\leq$, $=$, and $\geq$.

Categories are defined in a similar manner as sensitivities using the **category** statement. Unlike sensitivities, categories **are not** hierarchically related and **form a set**. The key relations between sets of categories X, Y are $X \subseteq Y$ and $X \supseteq Y$. The allowed security level combinations are defined using the **level** statement.

Security level of X specified as **s0:c0.c2-s7:c0.c2** defines the security levels:

- *low* part – labelled as l_X – **s0:c0.c2**
- *high* part – labelled as h_X – **s7:c0.c2**

Security levels form a lattice and can be compared using the dominance relationship that can be used in the **mlsconstrain** rules. Let $\text{sens}(SL)$ denote the sensitivity of the security level SL and $\text{cat}(SL)$ denote the set of categories of the security level SL. The dominance relation is defined as follows:

$$\begin{aligned} SL_1 \text{ dom } SL_2 \Leftrightarrow \quad & (\text{sens}(SL_1) \geq \text{sens}(SL_2)) \\ & \wedge (\text{cat}(SL_1) \supseteq \text{cat}(SL_2)) \\ SL_1 \text{ domby } SL_2 \Leftrightarrow \quad & (\text{sens}(SL_1) \leq \text{sens}(SL_2)) \\ & \wedge (\text{cat}(SL_1) \subseteq \text{cat}(SL_2)) \end{aligned}$$

The principal part of the rules of our policy for the read and write operations, that apply to unstrusted and trusted subjects (i.e. to all but partially trusted subjects), is based on comparisons of confidentiality and integrity levels of the subject and the object:

$$CR_S \geq C_O \wedge IR_S \leq I_O \text{ for read}(S,O)$$

and

$$CW_S \leq C_O \wedge IW_S \geq I_O \text{ for write}(S,O).$$

for write.

Noticing this, we decided to use the sensitivity to encode our model's confidentiality level, and categories to encode our model's integrity level. Untrusted and trusted subjects in our model have two confidentiality and two integrity levels assigned (for reading and for writing). We decided to use the low and high parts of the security level to encode the two pairs of confidentiality and integrity levels:

- *Low* part of security level will correspond to the attributes used for the **read** operation (C_O, I_O, and CR_S, IR_S, respectively)
- *High* part of security level will correspond to the attributes used for the **write** operation (C_O, I_O, and CW_S, IW_S, respectively).

A natural choice is to define a sensitivity level for each confidentiality level, and a category for each integrity level. Unfortunately, it appears that SELinux does not allow the low and high security levels of an object or of a subject to differ only in the categories (e.g. a security context with the security level range **s0:c0-s0:c1,c2** is considered **invalid**). We found a solution that uses two sensitivity levels to encode the same confidentiality level, depending on whether it is used in the low (read) or high (write) part. An object with $C_O = X$ and $I_O = Y$ is assigned the security context:
*system_u:system_r:obj_l:**sX:cY-s(X+5):cY***.

The principal part of the rules for the read operation can then be evaluated using the **dom** operator provided we find a suitable mapping between the integrity level IR_S and categories for the low security level of the subject. If we denote the low security level of a subject S as l_S and the low security level of an object O as l_O, we need to satisfy:

$$l_S \text{ dom } l_O \iff (CR_S \geq C_O \wedge IR_S \leq I_O)$$

That is

$$\text{sens}(l_S) \geq \text{sens}(l_O) \iff CR_S \geq C_O$$

and

$$\text{cat}(l_S) \supseteq \text{cat}(l_O) \iff IR_S \leq I_O$$

The principal part of the rules for the write operation can also be evaluated using the **dom** operator, if we can satisfy the following (where h_X denotes the high security level of X):

$$h_S \text{ dom } h_O \iff (CW_S \leq C_O \wedge IW_S \geq I_O)$$

That is

$$\text{sens}(h_S) \geq \text{sens}(h_O) \iff CW_S \leq C_O$$

and

$$\text{cat}(h_S) \supseteq \text{cat}(h_O) \iff IW_S \geq I_O$$

All of these are satisfied when we define 6 sensitivity levels $s0 < s1 < s2 < s7 < s6 < s5$ and three categories $c0, c1, c2$ and use the mappings given in the tables below to assign the sensitivity and the categories to the low and high security levels for subjects. The ordering of the sensitivity level can be easily specified using the SELinux statement: **dominance** { s0 s1 s2 s7 s6 s5 }.

Operations	read	write
confidentiality level	sensitivity *low*	sensitivity *high*
C_S0	s0	s5
C_S1	s1	s6
C_S2	s2	s7

Operations	read	write
Integrity	categories *low*	categories *high*
I_S0	c0.c2	c0
I_S1	c1.c2	c0.c1
I_S2	c2	c0.c2

The notation c0.c2 means categories c0,c1,c2.

The principal part of the read and write rules can be specified as simple constraints:

```
mlsconstrain { file } { read } (
 (l1 dom l2)
);

mlsconstrain { file } { write } (
 (h1 dom h2)
);
```

Where:

- l1 represents *low* security level of subject
- h1 represents *high* security level of subject
- l2 represents *low* security level of object
- h2 represents *high* security level of object

There are other parts of the rules for the operations, however. First, there are rules comparing the user identities assigned to the subject and the object with some exceptions, and then there are rules for partially trusted subjects. The former can be solved in a way similar to what we used in [12, 13] – we would just need a special DTE type attribute assigned to the object with $C_O = 2$ and/or $I_O = 2$. The part of the read and write rules, comparing the user identities assigned to the subject and the object, can be specified as simple constraints:

```
constrain { file } { read } (
(U1 == U2 or T2 != C2)
and
(U1 == U2 or T2 != I2 or U2 == {root system_u})
);

constrain { file } { write } (
(U1 == U2 or T2 != I2)
and
(U1 == U2 or T2 != C2 or U2 == {root system_u})
);
```

Partially trusted subjects can also be handled using specific DTE types for the subjects and specific DTE types for labelled objects in a way similar to that in [12, 13]. These constraints specified above would need to be extended to allow for an exception if the subject is one of a partially trusted type. Let's do this now.

In the best case, the complete rules for operation read(S,O) look like these:

```
mlsconstrain { file } { read } (
(l1 dom l2)
or
[ rules for partially trusted subjects ]
);

constrain { file } { read } (
(U1 == U2 or T2 != C2)
and
(U1 == U2 or T2 != I2 or U2 == {root system_u})
);
```

Where:

- l1 represents *low* security level of subject
- l2 represents *low* security level of object

However, the reality was different. The *partially trusted subject S* can use the substatements in the rules for operations read(S,O) and write(S,O) this way. *Subject S* **satisfies** the substatement of confidentiality for trusted or untrusted subjects and it **satisfies** the substatement of integrity for partially trusted subjects or vice versa. Thus, the use of mlsconstrain operations (dom or domby) on MLS SELinux is useless in this case. Thus, we did not develop this solution further. We denoted this solution as **MLS-1.0**.

4.3 Multi-Level Security Way-2.0

In this section, we present on our second attempt of implementation that uses MLS components of SELinux. The ideas are based on utilizing MLS components of SELinux in addition to the DTE component and modification of MLS SELinux operations.

```
static inline int mls_level_dom(const struct mls_level *l1,
const struct mls_level *l2)
{
 return ((l1->sens >= l2->sens)
        && ebitmap_contains(&l1->cat, &l2->cat));
}
```

Figure 4.1: Source code of MLS constraint operation dom

4.3.1 Solution

This solution is based on the MLS-1.0 (see previous section 4.2). The difference is in implementation of partially trusted subjects.

Firstly, we decided to disable the check of security levels dominance. For a security context to be **valid**, the *high* level **must always dominate** the *low* level. This condition is contained in SELinux policy compiler, SELinux kernel, and sepol library.

Next, we defined new mlsconstrain operations, namely *sdom* and *cdom*, respectively. The original mlsconstrain operation *dom* is defined as follows:

- **dom**: Security level SL1 *dom (dominates)* security level SL2 if the sensitivity of SL1 is higher or equal to the sensitivity of SL2, and the categories of SL1 are a superset of the categories of SL2.

This operation is declared in sepol library and kernel, namely *sepol/policydb/mls_types.h* and *selinux/ss/mls_types.h*, respectively. For source code see fig. 4.1. The new mlsconstrain operations *sdom* and *cdom* are defined as follows:

- **sdom**: Security level SL1 *sdom (sensitivity dominates)* security level SL2 if the sensitivity of SL1 is higher or equal to the sensitivity of SL2
- **cdom**: Security level SL1 *cdom (categories dominate)* security level SL2 if the categories of SL1 are a superset of the categories of SL2.

And therefor applies for these operations:

$$\text{dom(SL1, SL2)} \Leftrightarrow \text{sdom(SL1, SL2)} \wedge \text{cdom(SL1, SL2)}$$

For source code see fig. 4.2.
We also enabled these operations in SELinux policy compiler, namely checkpolicy compiler.

```
static inline int mls_level_sdom(const struct mls_level *l1,
 const struct mls_level *l2)
{
 return (l1->sens >= l2->sens);
}

static inline int mls_level_cdom(const struct mls_level *l1,
 const struct mls_level *l2)
{
 return (ebitmap_contains(&l1->cat, &l2->cat));
}
```

Figure 4.2: Source code of new MLS constraint operations sdom and cdom

Since we disabled the check of the security levels dominance, we **did not** need to use all sensitivities from MLS-1.0 solution. Now SELinux **allows** the low and high security levels of an object or a subject to differ only in the categories (e.g. a security context with the security level range **s0:c0-s0:c1,c2** is considered **valid**). An object with $C_O = X$ and $I_O = Y$ is assigned the security context:

*system_u:system_r:obj_l:**sX:cY-s(X+0):cY***

Thus, we defined only 3 sensitivity levels $s0 < s1 < s2$ and three categories $c0, c1, c2$ and used the mappings given in the tables below to assign the sensitivity and the categories to the low and high security levels for subjects. The ordering of the sensitivity level was easily specified as follows:
dominance { s0 s1 s2 }.

Operations	read	write
confidentiality level	sensitivity *low*	sensitivity *high*
C_S0	s0	s0
C_S1	s1	s1
C_S2	s2	s2

Operations	read	write
Integrity	categories *low*	categories *high*
I_S0	c0.c2	c0
I_S1	c1.c2	c0.c1
I_S2	c2	c0.c2

The notation c0.c2 means categories c0,c1,c2. The fig. 4.3 shows the resulting rules. We denoted this solution as **MLS-2.0**.

```
mlsconstrain { file } { read } (
 (l1 sdom l2)
or
(
((T2 == L0 and T1 == CRLS0) or (T2 == L1 and T1 == CRLS1))
 and
((T2 == C2 and T1 == CRL2) or (T2 == C1 and T1 == {CRL1 CRL2}))
)
);

mlsconstrain { file } { read } (
 (l1 cdom l2)
or
(
((T2 == L0 and T1 == IRLS0) or (T2 == L1 and T1 == IRLS1))
 and
(( T2 == I0 and T1 == IRL0) or (T2 == I1 and T1 == {IRL1 IRL0}))
)
);

constrain { file } { read } (
(U1 == U2 or T2 == {C0 C1})
and
(U1 == U2 or T2 == {I0 I1} or U2 == {root system_u})
);
```

Figure 4.3: New rules for read(S,O)

4.3.2 Comparison

Now, we compare new solution MLS-2.0 with old solution presented in our bachelor's thesis [13].

Complexity of Types

In both case, we use DTE types to encode the security attributes of objects and partially trusted subjects. Furthermore, we use MLS to encode the security attributes of trusted and untrusted subjects. The number of types for object and partially trusted subjects stay the same. We duplicate the security attributes of objects because each of objects must have

the DTE types part and MLS context part. We spare some types for subject theoretically, but nothing in real practise.

Complexity of Rules

The number of substatements in the rules for trusted and untrusted subjects is constant and the number of substatements in the rules for partially trusted subjects is linear.

Disadvantages

We would like to point out that there exist important disadvantages of our old solution. From the point of view of the new solution, we would like to check these disadvantages now:

1. Long expressions in the rules
2. The number of substatements in the rules is linear
3. Unique name for type for each of possible combinations of the attribute valuesy
4. TE transition rules for each of possible combinations of the subjects and objects that are used in *create* mechanism

The first, second, and third item have been partially solved. The fourth item has not been solved at all.

4.4 Multi-Level Security Way-3.0

In this section, we present on our third attempt of implementation that use MLS components of SELinux. The ideas are only based on utilizing MLS components of SELinux and modification of MLS SELinux operations.

4.4.1 Solution

This solution is based on the MLS-2.0 (see previous section 4.3). The difference is in implementation of partially trusted subjects.

Firstly, we decided to transform partially trusted subjects from DTE security context into MLS one. We found a suitable mapping between the attributes for the trusted and/or untrusted subjects and attributes for the partially trusted subjects. We decided to use

the sensitivity to encode the two confidentiality levels for all subjects. For operation read(S,O), we defined the following priority sequence:

- Rule: $CR_S \geq C_O \vee (CRL_S \geq C_O \wedge L_O \in CRLS_S)$
- Priority sequence: $00 \rightarrow 11 \rightarrow \underline{01} \rightarrow \mathbf{10} \rightarrow 22 \rightarrow \underline{02} \rightarrow \underline{12} \rightarrow \mathbf{21} \rightarrow \mathbf{20}$

For operation write(S,O), we defined the following priority sequence:

- Rule: $CW_S \leq C_O \vee (CWL_S \leq C_O \wedge L_O \in CWLS_S)$
- Priority sequence: $\mathbf{02} \rightarrow \mathbf{01} \rightarrow \underline{20} \rightarrow \underline{10} \rightarrow 00 \rightarrow \underline{21} \rightarrow 12 \rightarrow 11 \rightarrow 22$

Where:

- XY represents confidentiality level X for trusted and/or untrusted subject (CR_S or CW_S) and confidentiality level Y for partially trusted subject (CRL_S or CWL_S).
- **XY** is useless in context of appropriate rule
- $\underline{\text{XY}}$ needs special care

In other words, we transform each trusted and untrusted subject into partially trusted subject without any labels. These subjects have got same values of appropriate confidentialities, i.e. theirs sensitivities (*low* or *high*) have form XX, where X is the original confidentiality value. Underlined sensitivities need special care because we need to check that the partially trusted subject has got a proper label.

This would be a nice solution. We need to ensure that this concept satisfies rules for operation read (see 1.1a) and write (see 1.2a). To do this, we need a *selector* to select a concrete sensitivity and compare it with our given sensitivity value. Unfortunately, MLS SELinux does not provide this necessary selector and the implementation of this selector would be a bit complex, i.e. we have had to refactor all MLS SELinux. Thus, this solution is **useless**.

4.4.2 Strict Security Model

So, we chose a different approach. We looked at the conditions for untrusted subjects and partially trusted subjects (see 1.5 and 1.6). We replaced some inequalities with equalities. Hence, we got a **strict** security model. New conditions are described below:

Each untrusted subject S must satisfy:

$$CW_S = CWL_S = CR_S = CRL_S$$
$$IW_S = IWL_S = IR_S = IRL_S$$

Each partially trusted subject S must satisfy:

$$CR_S = CWL_S < CRL_S = CW_S$$
$$IR_S = IWL_S > IRL_S = IW_S$$

Next, we used the ideas from MLS-2.0 (see section 4.3). Thus, we decided to use the low and high parts of the security level to encode the two pairs of confidentiality and integrity levels for all objects and subjects:

- *Low* part of security level will correspond to the attributes used for the **read** operation (C_O, I_O, resp. $CR_S = CWL_S, IR_S = IWL_S$).
- *High* part of security level will correspond to the attributes used for the **write** operation (C_O, I_O, resp. $CW_S = CRL_S, IW_S = IRL_S$).

In other words, the *low* security context represents attributes of the operation *read* for trusted and untrusted subjects, and attributes of the operation *write* for partially trusted subject in the same time. On the other hand, the *high* security context represents attributes of the operation *write* for trusted and untrusted subjects, and attributes of the operation *read* for partially trusted subject in the same time too. We used our new operations (see subsection 4.3.1) to work with this concept. New constraints look like these:
Rule for operation *read(S,O)*:

- mlsconstrain { file } { read } (
  (l1 sdom l2 **or** (h1 sdom l2 + label_check))
  and
  (l1 cdom l2 **or** (h1 cdom l2 + label_check))
  and user_check);

Rule for operation *write(S,O)*:

- mlsconstrain { file } { write } (
  (l2 sdom h1 **or** (l2 sdom l1 + label_check))
  and
  (h1 cdom l2 **or** (l1 cdom l2 + label_check))
  and user_check);

Where:

- l1 represents *low* security level of subject
- h1 represents *high* security level of subject
- l2 represents *low* security level of object
- h2 represents *high* security level of object
- label_check represents check that the partially trusted subject has a proper label when it tries to access to object.
- user_check represents the rules 1.1c and 1.1d located on page 19.

New constraints look good. Unfortunately, there is small problem. SELinux (and SELinux policy compiler too) supports only the following context for mls operations (role_mls_op):

- l1 role_mls_op l2
- l1 role_mls_op h2
- h1 role_mls_op l2
- h1 role_mls_op h2
- l1 role_mls_op h1
- l2 role_mls_op h2

Where *role_mls_op* can represent operations eq, dom, domby, incomp, sdom, or cdom.

That means that SELinux can not compare the *low* security level of object (l2) and the *high* security level (h1) of subject. This corresponds with the first substatement in write(S,O), namely (l2 sdom h1). Further, that means that SELinux can not compare *low* security level of object (l2) and *low* security level (l1) of subject. This corresponds with the second substatement in write(S,O), namely (l2 sdom l1 + label_check).
Thus, we added these comparisons to the SELinux, namely SELinux policy compiler, SELinux kernel, and sepol library.

There is the corresponding source code:

```
#define CEXPR_L2H1 2048 /* low level 2 vs. high level 1 */
#define CEXPR_L2L1 4096 /* low level 2 vs. low level 1 */
```

There is the corresponding source code for compiler that allows us to declare these rules:

```
| L2 role_mls_op H1
{ $$ = define_cexpr(CEXPR_ATTR, CEXPR_L2H1, $2);
  if ($$ == 0) return -1; }
| L2 role_mls_op L1
{ $$ = define_cexpr(CEXPR_ATTR, CEXPR_L2L1, $2);
  if ($$ == 0) return -1; }
```

The last remaining difficulties are label_check and user_check. The user_check can be solved easily by comparing user's identities and declaring specific attributes *c_shareable* for confidentiality and *i_shareable* for integrity. Thus, the last difficulty is label_check. Our first idea was this one. We will encode all labels to the categories. At first sight this is possible because each of objects has only its label and each of subjects can have a set of labels. We need to compare that the object's label is in the set of subject's labels. For this comparison we can use operation *cdom*.

Unfortunately, there is a problem. We can not compare labels for confidentiality and integrity independently. To solve this problem, we decided to look at implementation of categories in Linux kernel. The main idea is a **division** of categories into three independent parts (see fig. 4.5). In this way, we will obtain new meaning for security contexts of our security levels.
For *low* security level:

- The first part of categories will represent attribute $IR_S = IWL_S$.
- The second part of categories will represent object's label CL or subject's set of labels for confidentiality $CWLS_S$.
- The last (third) part of categories will represent object's label IL or subject's set of labels for integrity $IWLS_S$.

Security level	
sensitivity	categories

Object low		
C_O	I_O	$CL + IL$

Subject low		
CR_S	IR_S	$CWLS_S + IWLS_S$
CWL_S	IWL_S	

Subject high		
CW_S	IW_S	$CRLS_S + IRLS_S$
CRL_S	IRL_S	

Figure 4.4: Visualization of new security levels without divided categories

For *high* security level:

- The first part of categories will represent attribute $IW_S = IRL_S$.
- The second part of categories will represent subject's set of labels for confidentiality $CRLS_S$.
- The last (third) part of categories will represent subject's set of labels for integrity $IRLS_S$.

The categories are implemented as a vector of bits in SELinux kernel. More detailed information about implementation of categories in SELinux can be found in next section 4.5.

4.4.3 Rules

So, we have the good idea and now, we transform it into reality. We can use the mls operation *sdom* to work with sensitivities, but we can not use the mls operation *cdom* to work with categories. Thus, we replace the operation *cdom* with a set of operations that enable us to work with the divided categories.

Security level			
sensitivity	1.part	2.part	3.part

Object low			
C_O	I_O	CL	IL

Subject low			
CR_S	IR_S	$CWLS_S$	$IWLS_S$
CWL_S	IWL_S		

Subject high			
CW_S	IW_S	$CRLS_S$	$IRLS_S$
CRL_S	IRL_S		

Figure 4.5: Visualization of new security contexts of our security levels with divided categories

- Firstly, we defined a new operation ***icat*** for work with the first part of categories, namely with attributes $IR_S = IWL_S$ and $IW_S = IRL_S$.
- Secondly, we defined operation ***clcat*** for work with the second part of categories, namely with labels for confidentiality $CWLS_S$ and $CRLS_S$.
- Finally, we defined operation ***ilcat*** for work with the third part of categories, namely with labels for interity $IWLS_S$ and $IRLS_S$.

Source code for these operations:

```
static inline int mls_level_icat(const struct mls_level *l1,
const struct mls_level *l2){
return (ebitmap_contains_icat(&l1->cat, &l2->cat));}
static inline int mls_level_clcat(const struct mls_level *l1,
const struct mls_level *l2){
return (ebitmap_contains_clcat(&l1->cat, &l2->cat));}

static inline int mls_level_ilcat(const struct mls_level *l1,
const struct mls_level *l2){
return (ebitmap_contains_ilcat(&l1->cat, &l2->cat));}
```

Where functions *ebitmap_contains_**(struct ebitmap *e1, struct ebitmap *e2)* were created by modifying the original function *int ebitmap_contains(struct ebitmap *e1, struct ebitmap *e2)*.

4.4.4 Constraints

Now we have all means for rewriting rules from subsection 4.4.2. These rewritten rules are described below:
Rule for operation *read(S,O)*:

- mlsconstrain { file } { read } (
  (l1 sdom l2 *or* (h1 sdom l2 and h1 clcat l2))
  and
  (l1 icat l2 *or* (h1 icat l2 and h1 ilcat l2))
  and user_check);

Rule for operation *write(S,O)*:

- mlsconstrain { file } { read } (
  (l2 sdom h1 *or* (l2 sdom l1 and l1 clcat l2))
  and
  (h1 icat l2 *or* (l1 icat l2 and l1 ilcat l2))
  and user_check);

We can easily improve these rules. Firstly, we switch the second part of vector of categories with the third part of vector of categories. Fig. 4.6 shows this change. Then, we have to redefine the operation *ilcat* so that it would works like the operation *icat* and the original operation *ilcat* together, i.e. we control the attribute for integrity level and the appropriate label in the same time.
That means that we can replace the substatement *(l1 icat l2 or (h1 icat l2 and h1 ilcat l2))* with this one *(l1 icat l2 or (h1 ilcat l2))*. The updated rules are described below.
Modified rule for operation *read(S,O)*:

- mlsconstrain { file } { read } (
  (l1 sdom l2 *or* (h1 sdom l2 and h1 clcat l2))
  and
  (l1 icat l2 *or* (h1 ilcat l2))
  and user_check);

Security level			
sensitivity	1.part	2.part	3.part

Object low			
C_O	I_O	IL	CL

Subject low			
CR_S	IR_S	$IWLS_S$	$CWLS_S$
CWL_S	IWL_S		

Subject high			
CW_S	IW_S	$IRLS_S$	$CRLS_S$
CRL_S	IRL_S		

Figure 4.6: Visualization of new security levels with modified divided categories

Modified rule for operation *write(S,O)*:

- mlsconstrain { file } { read } (
  (l2 sdom h1 *or* (l2 sdom l1 and l1 clcat l2))
  and
  (h1 icat l2 *or* (l1 ilcat l2))
  and user_check);

We denoted this solution as **MLS-3.0**.

4.4.5 Comparison

Now, we compare this solution MLS-3.0 with old solution presented in our bachelor's thesis [13].

Complexity of Types

In this solution, we **do not** use DTE types to encode the security attributes of objects and partially trusted subjects. We use only MLS to encode all security attributes of trusted, untrusted, and partially trusted subjects. We need two types as a solution for

user_check, where we compare user's identities and evaluate the condition for specific attributes *c_shareable* for confidentiality and *i_shareable* for integrity.

Complexity of Rules

The number of substatements in the rules for all trusted, untrusted, and partially trusted subjects is **constant**.

Disadvantages

Now, we look at important disadvantages of our old solution. From the viewpoint of this solution, we would like to check these disadvantages:

1. Long expressions in the rules
2. The number of substatements in the rules is linear
3. Unique name for type for each of possible combinations of the attribute values
4. TE transition rules for each of possible combinations of the subjects and objects that are used in *create* mechanism

The first, second, and third item have been **fully solved**. The fourth item **has not been solved at all** because we only replace the TE transition rules with the MLS range_transition rules like in previous solution 4.3. Unfortunately, there is **one new disadvantage** called *Strict security model*. We can analyze this solution but we do not have to. The reason for this is that we will reuse all ideas that were presented in sections on previous pages. We present the effective use of these ideas in next section.

4.5 Multi-Category Security WAY-4.0

In this section, we describe our final solution. In this solution, we use all good ideas that were presented in previous sections.

4.5.1 Solution

This solution is based on the solution MLS-3.0 and *Strict security model* (see previous section 4.4). We will implement the whole non-strict security model, not only its strict version that was presented in previous section (see subsection 4.4.2). Thus, we **do not**

have to divide the vector of bits of categories into three parts. We **will encode all** attributes of model into independent parts of the vector of categories. We denoted this solution as **MCS-4.0**.

4.5.2 Implementation of Categories in SELinux

In this subsection, as we promised in previous section, we describe in details implementation of categories in the SELinux kernel.

Implementation of categories is contained in a file */security/selinux/ss/ebitmap.h*. Categories are implemented as an extensible bitmap. *An extensible bitmap* is a bitmap that supports an arbitrary number of bits. The extensible bitmaps are used to represent sets of values, such as types, roles, categories, and classes. Each extensible bitmap is implemented as a linked list of bitmap nodes, where each bitmap node has an explicitly specified starting bit position within the total bitmap.

Categories are implemented in a structure called *ebitmap* that consists of a highest bit and a structure called *ebitmap_node*. The structure *ebitmap_node* contains a link to next *ebitmap_node*, a starting bit, and something called *unsigned long maps[EBITMAP_UNIT _NUMS]*. These *maps* are very important for us. The *maps* are an array of 32 bits. The length of the array is defined as a value of *EBITMAP_UNIT_NUMS*. This value is defined as a complex expression *(32 - sizeof(void *) - sizeof(u32)) / sizeof(unsigned long)*, where value of:

- *sizeof(void *)* is 4,
- *sizeof(u32)* is 4,
- *sizeof(unsigned long)* is 4.

Thus, the length of the array *maps* is 6, because (32 -4 -4) / 4 is 6. Fig. 4.7 shows source code.

Summary

Each node contains 6 x *maps[i]*, where each of *maps[i]* contains 32 bits. Thus, we have 192 bits per node. Each of these bits represents a separate category.

```
#define EBITMAP_UNIT_NUMS  ((32 - sizeof(void *) - sizeof(u32)) \
                            / sizeof(unsigned long))

struct ebitmap_node {
       struct ebitmap_node *next;
       unsigned long maps[EBITMAP_UNIT_NUMS];
       u32 startbit;
};

struct ebitmap {
       struct ebitmap_node *node;   /* first node in the bitmap */
       u32 highbit;    /* highest position in the total bitmap */
};
```

Figure 4.7: Source code of implementation of categories

4.5.3 Non-Strict Security Model

We **will encode all** attributes of model into independent parts of the vector of categories. Thus we have to encode object's attributes and subject's attributes into this vector of categories. We chose a *block of 8 bits* as a natural delimiter, that means we have 8 levels for confidentiality and integrity. Now we describe our solution.

Node and Maps

Now we look at the first *maps[0]* in the first node. The attributes C_O,CR_S, and CW_S are implemented in the first *block[0]* of the *maps[0]*. The second *block[1]* contains the attributes I_O,IR_S, and IW_S. The third *block[2]* contains the attributes CRL_S and CWL_S. The fourth *block[3]* contains the attributes IRL_S and IWL_S.

So, we have done all baseline attributes for levels of the confidentiality and integrity. The last remaining attributes are labels of objects and the labels for confidentiality and integrity of subjects. Thus, the *maps[1]* and *maps[2]* contain the labels for $CRLS_S$ and $CWLS_S$ in the first node. Of course, we have to have the same number of labels for confidentiality and integrity. We actually have 64 labels for the confidentiality. Thus, the *maps[3]* and *maps[4]* have to contain only the labels for $IRLS_S$ and $IWLS_S$. Source codes are showed in fig. 4.8 and 4.9. Fig. 4.10 shows the visualization of the first maps[0].
Note: The position of category in vector of bits of categories is given by the position of declaration of this category in source file of policy for SELinux, namely *policy.conf*.

```
# Each node contains 6 items of maps ([0]-[5])
# Each maps contains 32 bits

# Confidentiality
category c0;    # confidentiality 0 - public data
category c1;    # confidentiality 1 - C-normal
category c2;    # confidentiality 2 - C-sensitive
# Reserved categories
category c3;
...
category c7;

# Integrity
category i0;    # integrity 0 - potentially malicious data
category i1;    # integrity 1 - I-normal
category i2;    # integrity 2 - I-sensitive
#  Reserved categories
category i3;
...
category i7;
```

Figure 4.8: Declaration of categories in the policy - part 1

4.5.4 Functions for Work with Categories

Functions for work with categories are defined in a file */security/selinux/ss/ebitmap.c*. As might be expected these functions work on all bits in vector of categories. Thus, we defined new operations that replace our previous declared operations *sdom* and *cdom* (see subsection 4.3.1). As you see later, these new operations will cover the whole operations *read(S,O)* and *write(S,O)*.
New operations (role_mls_op) are these:

- *subject* rule_read *object*
- *subject* rule_write *object*
- *subject* cshare *object*
- *subject* ishare *object*

Now we express these functions in the view of attributes of security model. Let do this.

The operation *subject* rule_read *object* covers the conditions:

$$[CR_S \geq C_O \vee (CRL_S \geq C_O \wedge L_O \in CRLS_S)] \wedge [IR_S \leq I_O \vee (IRL_S \leq I_O \wedge L_O \in IRLS_S)]$$

```
# Confidentiality labelled level
category clS0;    # confidentiality 0 - public data
category clS1;    # confidentiality 1 - C-normal
category clS2;    # confidentiality 2 - C-sensitive
#  Reserved categories
category clS3;
...
category clS7;

# Integrity labelled level
category ilS0;    # integrity 0 - potentially malicious data
category ilS1;    # integrity 1 - I-normal
category ilS2;    # integrity 2 - I-sensitive
#  Reserved categories
category ilS3;
...
category ilS7;

# Confidentiality labels
category cL0;
...
category cL63;

# Integrity labels categories
category il0;
...
category il63;
category ps; # Technical detail
category sentinel; # Technical detail
```

Figure 4.9: Declaration of categories in the policy - part 2

The operation *subject* rule_write *object* covers the conditions:

$$[CW_S \leq C_O \vee (CWL_S \leq C_O \wedge L_O \in CWLS_S)] \wedge [IW_S \geq I_O \vee (IWL_S \geq I_O \wedge L_O \in IWLS_S)]$$

The operation *subject* cshare *object* covers the condition:

$$C_O \leq 1$$

The operation *subject* ishare *object* covers the condition:

$$I_O \leq 1$$

Fig. 4.11 shows source code of these operations for compiler of SELinux policy.

The First Maps[0]

block[0]	c0	c1	c2	c3	c4	c5	c6	c7
block[1]	i0	i1	i2	i3	i4	i5	i6	i7
block[2]	clS0	clS1	clS2	clS3	clS4	clS5	clS6	clS7
block[3]	ilS0	ilS1	ilS2	ilS3	ilS4	ilS5	ilS6	ilS7

Figure 4.10: Visualization of the first maps[0]

```
#define CEXPR_CSHARE    6
#define CEXPR_ISHARE    7
#define CEXPR_READ    8
#define CEXPR_WRITE   9

case CEXPR_READ:
                s[++sp] = mls_level_rule(l1, l2);
                continue;
case CEXPR_WRITE:
                s[++sp] = mls_level_rule(l1, l2);
                continue;
case CEXPR_CSHARE:
                s[++sp] = mls_level_cshare(l1, l2);
                continue;
case CEXPR_ISHARE:
                s[++sp] = mls_level_ishare(l1, l2);
                continue;
```

Figure 4.11: Declaration for compiler

4.5.5 Functions and Rules

From the viewpoint of the implementation, the rules *read (S,O)* and *write (S,O)* are identical. That means that they can be expressed using the same function, namely *mls_level_rule()* (see fig. 4.11), because the security contexts for *write* and *read* are linked to the *low* security level and *high* security level, respectively.

Function *mls_level_rule()* contains our new functions that we added into SELinux kernel, namely in a file */security/selinux/ss/ebitmap.c*. These functions work with the independent parts of teh vector of categories. They check that the categories of subject are a superset of the categories of object except empty set of course. Source code of function *mls_level_rule()* is described in fig. 4.12. In previous solution 4.4, we used two special attributes in the *user_check* part of rule. Now we do not have to use these special attributes because we have specialized operation for this purpose, namely *cshare*

```
static inline int mls_level_rule(const struct mls_level *l1,
const struct mls_level *l2){
return ((ebitmap_contains_cdom(&l1->cat, &l2->cat) ||
         ebitmap_contains_cldom(&l1->cat, &l2->cat))
&&
        (ebitmap_contains_idom(&l1->cat, &l2->cat) ||
         ebitmap_contains_ildom(&l1->cat, &l2->cat)));
}
```

Figure 4.12: Function *mls_level_rule()* - source code

```
mlsconstrain { file } { read } (
        (h1 rule_read l2) and (
        (U1 == U2) or ( h1 cshare l2 and
        (h1 ishare l2 or U2 == {root system_u}) )
        )
);

mlsconstrain { file } { write } (
        (l1 rule_write l2) and (
        (U1 == U2) or ( l1 ishare l2 and
        (l1 cshare l2 or U2 == {root system_u}) )
        )
);
```

Figure 4.13: Final full version of *read(S,O)* and *write(S,O)*

and *ishare*.

Fig. 4.13 shows final full version of the rules, namely *read(S,O)* and *write(S,O)*.

Note: In this time, SELinux does not allow to declare a substatement *U2 == {root system_u}*, i.e. a SELinux compiler does not recognize these user's identities. That means that the SELinux compiler does not satisfy the specification for the SELinux mlsconstraints. This may be fixed in the new version of the compiler. Thus, we replaced the substatement *U2 == {root system_u}* with the substatement *U1 != U2*. This is a temporary solution until the compiler is corrected.

4.5.6 Security Context

In this subsection, we describe how to set a proper security context to the objects and subjects. We use the mappings given in the following tables to assign the categories to the *low* and *high* security levels for objects and subjects. In comparison with the previous

solution MLS-3.0 in section 4.4, we swap the meaning of *low* and *high* security levels, i.e. *low* and *high* security levels represent security context of *write* and security context of *read*, respectively.

	Security context	**write**	**read**
	Security level	*low*	*high*
Value of Object's attributes	$C_O = 0$	c0	–
	$C_O = 1$	c1	–
	$C_O = 2$	c2	–
	$C_O = 3$	c3	–
	$C_O = 4$	c4	–
	$C_O = 5$	c5	–
	$C_O = 6$	c6	–
	$C_O = 7$	c7	–

	Security context	**write**	**read**
	Security level	*low*	*high*
Value of Object's attributes	$I_O = 0$	i0	–
	$I_O = 1$	i1	–
	$I_O = 2$	i2	–
	$I_O = 3$	i3	–
	$I_O = 4$	i4	–
	$I_O = 5$	i5	–
	$I_O = 6$	i6	–
	$I_O = 7$	i7	–

Setting the security context for objects

	Security context	**write**	**read**
	Security level	*low*	*high*
Value of Subject's attributes	$C_S = 0$	c0.c_{max}	c0
	$C_S = 1$	c1.c_{max}	c0.c1
	$C_S = 2$	c2.c_{max}	c0.c2
	$C_S = 3$	c3.c_{max}	c0.c3
	$C_S = 4$	c4.c_{max}	c0.c4
	$C_S = 5$	c5.c_{max}	c0.c5
	$C_S = 6$	c6.c_{max}	c0.c6
	$C_S = 7$	c7	c0.c7

	Security context	**write**	**read**
	Security level	*low*	*high*
Value of Subject's attributes	$CL_S = 0$	clS0.clS_{max}	clS0
	$CL_S = 1$	clS1.clS_{max}	clS0.clS1
	$CL_S = 2$	clS2.clS_{max}	clS0.clS2
	$CL_S = 3$	clS3.clS_{max}	clS0.clS3
	$CL_S = 4$	clS4.clS_{max}	clS0.clS4
	$CL_S = 5$	clS5.clS_{max}	clS0.clS5
	$CL_S = 6$	clS6.clS_{max}	clS0.clS6
	$CL_S = 7$	clS7	clS0.clS7

	Security context	**write**	**read**
	Security level	*low*	*high*
Value of Subject's attributes	$I_S = 0$	i0	i0.i_{max}
	$I_S = 1$	i0.i1	i1.i_{max}
	$I_S = 2$	i0.i2	i2.i_{max}
	$I_S = 3$	i0.i3	i3.i_{max}
	$I_S = 4$	i0.i4	i4.i_{max}
	$I_S = 5$	i0.i5	i5.i_{max}
	$I_S = 6$	i0.i6	i6.i_{max}
	$I_S = 7$	i0.i7	i7

	Security context	**write**	**read**
	Security level	*low*	*high*
Value of Subject's attributes	$IL_S = 0$	ilS0	ilS0.ilS_{max}
	$IL_S = 1$	ilS0.ilS1	ilS1.ilS_{max}
	$IL_S = 2$	ilS0.ilS2	ilS2.ilS_{max}
	$IL_S = 3$	ilS0.ilS3	ilS3.ilS_{max}
	$IL_S = 4$	ilS0.ilS4	ilS4.ilS_{max}
	$IL_S = 5$	ilS0.ilS5	ilS5.ilS_{max}
	$IL_S = 6$	ilS0.ilS6	ilS6.ilS_{max}
	$IL_S = 7$	ilS0.ilS7	ilS7

Setting the security context for subjects

Note: The notation c0.c_{max} means categories c0,c1, ..., c_{max}, similarly for the other categories. Attribute $attr_{max}$ denotes a maximal level that we will use in our operating system, i.e. when we use 3 levels for confidentiality and integrity (0, 1, and 2), the attribute $attr_{max}$ will represent the attribute $attr2$.

Our general implementation allows us to use maximal 8 levels for confidentiality and integrity. Our implementaion also support 64 labels. We will use the categories for labels this way:
When we assign a label $L4$ to an object, the security context of this object will have the categories $cL4$ and $iL4$ in *low* security level.
In case of subjects, the situation is different. When we assign a label $L4$ for attributes $CRLS_S$ and $IWLS_S$ and a label $L5$ for attribute $CRLS_S$ to a subject, the security context of this subject will have the category $iL4$ in the *low* security level and the categories $cL4$ and $cL5$ in the *high* security level.

4.5.7 Technical Details

Sensitivity

We have to define only one sensitivity level, namely $s0$, although we do not use it in this solution, i.e. it is useless for our solution. The ordering of the sensitivity level can be specified using the standard SELinux statement:
dominance { s0 }.

Sentinels

The categories *ps* and *sentinel* are our last declared categories. We use the category *sentinel* in declaration of security level. The category *ps* is used in *create mechanism*. The level statement enables the previously declared sensitivity and category identifiers to be combined into a security level:
level s0:c0.*sentinel*; .

Create Mechanism

SELinux security context of a newly created object is determined by special rules or can be automatically inherited from a parent subject that creates this new object. There is a corresponding function for this purpose in SELinux kernel, namely *int mls_compute_sid(struct context *scontext, struct context *tcontext, u16 tclass, u32 specified, struct context *newcontext, bool sock)* . This function, located in *security/selinux/ss/mls.c*, works as a standard create mechanism. It calls a function *mls_context_cpy_low(newcontext, scontext)*, located in a file */security/selinux/ss/context.h*, that sets both levels in the MLS range of 'dst' to the *low* level of 'src'. That means that the subject's *low* security level is copied into the *low* and *high* object's security level. For this purpose, it uses another function *ebitmap_cpy(&dst->range.level[1].cat, &src->range.level[0].cat)* that is located in a file */security/selinux/ss/ebitmap.c*.

We think that using this function is useless for us. Thus, we modified this automatic mechanism. We replaced the function *mls_context_cpy_low(newcontext, scontext)* with a new function *mls_context_cpy_low_new(newcontext, scontext, tcontext)* in the function *mls_compute_sid(...)*.

Our new *create mechanism* works like the original function. In addition, it converts the subject's *low* security level into the proper object's security level. That means that special function modifies only our reserved parts of the vector of categories that belong to the created object. The function transforms a set of subject's categories into appropriate object's category.

The function contains a body of original function, but we have added another function, namely *ebitmap_cpy_convert_context(...)*. This function ensures the transformation mentioned above. As you can see above, the new function **has a new parameter** *tcontext*. The parameter *tcontext* is a security context of parent directory. We need the security context of parent directory **to set a proper label** for the created file.

To ensure a correct behaviour of operation *create*, all subjects will have own appropriate categories for corresponding attributes and a special category *ps*. A subject will have the special category *ps* in its *low* security context. This category provides us the right length of vector of categories that we will modify in *create mechanism*.

On the other hand, if we **use specific rules** to determine *create mechanism*, we will need to use the *range_transition* rules (see subsection 2.1.5). The *range_transition* statement was enhanced in Policy version 21 (and greater) to accept other object classes. Thus, we can use the classes like dir, file, etc. in these rules. But these rules work with types. Therefore, we would have to declare an independent type for each of possible combinations of the attribute values (MLS categories). Thus, the use of the range_transition rules for *create mechanism* is **useless** for us.

Modification of Standard MLS SELinux Functions

To ensure a full separation from standard MLS SELinux functions, we modified some SELinux MLS functions. Thus, we defined new versions of the main functions that work with MLS security levels, except for defined independent parts of vector of categories. In this way, **we ensure backward compatibility**. That means that this step enables us to integrate our model into an arbitrary SELinux policy independently of the given policy. For detailed information about integration into the arbitrary SELinux policy, please see section 5.3.

4.5.8 Comparison

Now, we compare this solution MCS-4.0 with previous solution MLS-3.0.

Complexity of Types

In this solution, same as before, we **do not** use DTE types to encode the security attributes of objects and partially trusted subjects. We use only MLS to encode all security attributes of trusted, untrusted, and partially trusted subjects. Unlike the previous solution, we **do not** need two attributes like solution to user_check because we have created the specialized functions for this purpose.

Complexity of Rules

Same as before, the number of substatements in the rules for all trusted, untrusted, and partially trusted subjects is **constant**.

Disadvantages

Now, we look at important disadvantages of our old solution again. From the point of view of this solution, we would like to check the disadvantages of our previous solution presented in our bachelor's thesis [13].

1. Long expressions in the rules
2. The number of substatements in the rules is linear
3. Unique name for type for each of possible combinations of the attribute values
4. TE transition rules for each of possible combinations of the subjects and objects that are used in *create* mechanism

Same as before, the first, second, and third item have been **fully solved**. Unlike the previous solution, the fourth item **has been solved as well**.

Chapter 5

Testing

In this chapter, we present a practical impact of our work. In section 5.1, we wrote a step-by-step tutorial on installing our solution. In section 5.2, we wrote a tutorial on building a custom kernel with our solution. In section 5.3, we provide instructions for adjustment of arbitrary SELinux policy that this policy will contain our model.

5.1 Installation of our Solution

In this section, we provide instructions for advanced users who want to install our solution on their system. The installation of our solution consists of these steps:

1. Disabling SELinux
2. Compiling Modified SELinux Policy Compiler
3. Compiling SELinux Policy
4. Kernel installation
5. Enabling SElinux

The details of each step are described in the following subsections.

5.1.1 How to Disable SElinux

First of all, you need to disable SELinux. Disabling will completely disable all SELinux functions including file and process labelling. To permanently disable SElinux you need

to edit SElinux's config file */etc/selinux/config* and add/alter the following line to disable it:

```
SELINUX=disabled
```

5.1.2 Compiling Modified SELinux Policy Compiler

As you can see on previous pages, we made some important changes in SELinux kernel. Of course, these changes have impact on SELinux policy compiler, called *checkpolicy*, and *sepol* library. Thus, we modified the original compiler. We named new SELinux policy compiler as *mcheckpolicy*. Because *checkpolicy* (and *mcheckpolicy* too) uses *sepol* library, we modified it too. This modified library reflects our changes in SELinux kernel. New compiler can be compiled using the command *make*. The source files of *mcheckpolicy* are stored in *./public-MCS-4.0/checkpolicy-2.0.19-modif-4.0* and the source files of *sepol* library are stored in *./public-MCS-4.0/sepol*. You can use the following commands to compile *mcheckpolicy*:

```
cd ./public-MCS-4.0
cp sepol /usr/include
cd ./public-MCS-4.0/checkpolicy-2.0.19-modif-4.0
make
cp ./checkpolicy /usr/bin/mcheckpolicy
cp ./checkmodule /usr/bin/mcheckmodule
```

Note: If you use our solution with the SELinux reference policy (*refpolicy*), you do not need to rename *checkpolicy* and *checkmodule* but you can only copy them into */usr/bin*. You can use the following simple commands to compile *mcheckpolicy* as the standard *checkpolicy*:

```
cd ./public-MCS-4.0/checkpolicy-2.0.19-modif-4.0
make install
```

5.1.3 Compiling SELinux Policy

This subsection covers the considerations and methods for compiling SELinux policy. When you install a new policy, you must eventually reboot to test that it works during system start-up. If the policy change is significant enough, such as installing an entirely new policy, you need to reboot to ensure all applications are running in the right context for the loaded policy.

We provide a *minimalist policy* that demonstrates the possibilities of our implementation. This *minimalist policy* can be found in *./public-MCS-4.0/minimalist policy*. Thus, you can use the following commands to compile this policy:

```
cp ./public-MCS-4.0/minimalist policy /etc/selinux/minpolicy/
cd /etc/selinux/minpolicy/
mcheckpolicy -M -o ./policy/policy.24 ./src/policy.conf
```

Note: For compiling the SELinux reference policy that includes our model too, see section 5.3.

5.1.4 Kernel Patch and Kernel Installation

Finally, you need only compile the kernel with our patch. We have devoted a separate section to the kernel patch and the installation of modified kernel. Please see section 5.2.

5.1.5 How to Enable SELinux

If all is done, you need to enable SELinux. Enabling will completely enable all SELinux functions including file and process labelling. To enable SELinux you need to edit the SELinux's configuration file */etc/selinux/config* again and set SELinux into permissive mode:

```
SELINUX=permissive
```

After booting into permissive mode, run this command to relabel everything:

```
fixfiles relabel
```

Alternatively, in Fedora and RedHat Enterprise Linux you can run

```
touch /.autorelabel
```

and reboot.
Note that this can take quite some time for systems with a large number of files.

After relabelling the filesystem, you can switch to enforcing mode and your system should be fully enforcing again. Thus, you need to edit the file */etc/selinux/config* once again and only set SELinux into enforcing mode:

```
SELINUX=enforcing
```

5.2 Kernel Patch Installation

In this section, we provide instructions for advanced users who want to rebuild their kernel with our patch.
Note: However, when building or running any such kernel, you are pretty much on your own here if something does not work as you would hope or expect. These instructions apply to Fedora 12 and later releases.

5.2.1 Requirement

Before starting, make sure the system has all the necessary packages installed, including the following:

- rpmdevtools
- yum-utils

yum-utils is a default package. To install the other package, use the following command:

```
# su -c 'yum install rpmdevtools yum-utils'
```

5.2.2 Get the Source

1. Prepare a RPM package building environment in your home directory. Run the following command:

   ```
   # rpmdev-setuptree
   ```

   This command creates different directories $HOME/rpmbuild/SOURCES, $HOME/rpmbuild/SPECS, and $HOME/rpmbuild/BUILD. Where $HOME is your home directory.

2. Download the kernel-<version>.src.rpm file. Enable the appropriate source repositories with the –enablerepo switch. (yumdownloader –enablerepo=repo_to_enable –source kernel)

   ```
   # yumdownloader --source kernel
   ```

3. Install build dependencies for the kernel source with the yum-builddep command (root is required to install these packages):

```
# su -c 'yum-builddep kernel-<version>.src.rpm'
```

4. Install kernel-<version>.src.rpm with the following command:

```
# rpm -Uvh kernel-<version>.src.rpm
```

This command writes the RPM contents into $HOME/rpmbuild/SOURCES and $HOME/rpmbuild/SPECS, where $HOME is your home directory. It is safe to ignore any messages similar to the following:

```
warning: user kojibuilder does not exist - using root
warning: group kojibuilder does not exist - using root
```

Note: **Space Required**. The full kernel building process requires several gigabytes of extra space on the file system containing your home directory.

5.2.3 Prepare the Kernel Source Tree

This step expands all of the source code files for the kernel. This is required to view the code, edit the code, or to generate a patch.

1. Prepare the kernel source tree using the following commands:

```
# cd ~/rpmbuild/SPECS
# rpmbuild -bp --target=$(uname -m) kernel.spec
```

The kernel source tree is now located in the ~/rpmbuild/BUILD/kernel-<version>/ /linux-<version>.<arch> directory.

5.2.4 Prepare Build Files

This step makes the necessary changes to the kernel.spec file. This step is required for building a custom kernel.

1. Change to the /rpmbuild/SPECS directory:

   ```
   # cd ~/rpmbuild/SPECS
   ```

2. Open the kernel.spec file for editing.

3. Give the kernel a unique name. This is important to ensure the custom kernel is not confused with any released kernel. Add a unique string to the kernel name by changing the *buildid* line. Optionally, change ".local" to your initials, a bug number, the date or any other unique string.

   Change this line:

   ```
   #% define buildid .local
   ```

   To this (note the extra space is removed in addition to the pound sign):

   ```
   %define buildid .<custom_text>
   ```

4. If you generated a patch, add the patch to the kernel.spec file, preferably at the end of all the existing patches and clearly commented.

   ```
   # cputime accounting is broken, revert to 2.6.22 version
   Patch2220: linux-2.6-cputime-fix-accounting.patch
   Patch9999: linux-2.6-samfw-test.patch
   ```

   Our patch has name linux-2.6-modif-selinux.patch, located in *./public-MCS-4.0/*. Thus, you add only:

   ```
   Patch15555: linux-2.6-modif-selinux.patch
   ```

The patch then needs to be applied in the patch application section of the spec file. Again, at the end of the existing patch applications and clearly commented.

```
ApplyPatch linux-2.6-cputime-fix-accounting.patch

ApplyPatch linux-2.6-samfw-test.patch

ApplyPatch linux-2.6-modif-selinux.patch
```

Note: The patch needs to be copied into *./rpmbuild/SOURCES*.

5.2.5 Build the New Kernel

This step actually generates the kernel RPM files. This step is required for building a custom kernel with our patch.
Use the rpmbuild utility to build the new kernel:

1. To build with firmware included, do:

```
# rpmbuild -bb --with baseonly --with firmware

--without debuginfo --target=`uname -m` kernel.spec
```

The build process takes a long time to complete. A lot of messages will be printed to the screen. These messages can be ignored, unless the build ends with an error. If the build completes successfully, the new kernel packages will be located in the ~/rpmbuild/RPMS directory.

5.2.6 Install the New Kernel

Make sure that the SELinux is **disabled**. This step actually installs the new kernel into the running system. To install the new kernel, use the rpm -ivh command, not the -U or –upgrade options:

```
su -c "rpm -ivh
```

```
$HOME/rpmbuild/RPMS/<arch>/kernel-<version>.<arch>.rpm

$HOME/rpmbuild/RPMS/<arch>/kernel-firmware-<version>.<arch>.rpm

$HOME/rpmbuild/RPMS/<arch>/kernel-headers-<version>.<arch>.rpm

$HOME/rpmbuild/RPMS/<arch>/kernel-devel-<version>.<arch>.rpm"
```

These commands will install your kernel in /boot, create a new initramfs to bootstrap your kernel, and automatically add your new kernel to your grub bootloader "menu.lst". At this point, you can reboot to give control to your new kernel.

5.3 Adjustment of Arbitrary Policy

In this section, we provide instructions for adjustment of arbitrary SELinux policy so that this policy will contain our model.

An important result for us is the combination of an arbitrary SELinux policy with our solution. There are two types of SELinux policy:

1. SELinux policy does not use SELinux MLS context,
2. SELinux policy uses SELinux MLS context.

5.3.1 SELinux Policy without MLS

In the first case, we need to add and enable SELinux MLS in current policy. Consider we have a *monolithic* policy. A *monolithic* policy is an SELinux policy that is compiled from one source file called *policy.conf* (i.e. it does not use the Loadable Module Policy statements and infrastructure which therefore makes it suitable for embedded systems as there is no policy store overhead). An example monolithic policy is the NSAs original Example Policy.

A policy configuration file consists of the following top-level components:

1. FLASK definitions
2. **MLS declarations and MLS Constraint declarations**
3. Policy capability definitions

4. Type Enforcement (TE) and Role-Based Access Control (RBAC) declarations and rules
5. User declarations
6. Constraint declarations
7. Security context specification

First of all, we need to declare all necessary things that are used in our solution MCS-4.0 (see section 4.5). For this purpose, we have to only add content of file *./public-MCS-4.0/policy-src/MLS declaration* into the appropriate place in *policy.conf*. As you can see above, the appropriate place in *policy.conf* is between the *FLASK definitions* and the *Policy capability definitions* that are followed by *Type Enforcement (TE) and Role-Based Access Control (RBAC) declarations and rules*.

Next up, we have to add the mls context for user into *User declarations*. The modified users' declarations would look like these one:

```
user root roles { set of user's roles } \
\ level s0 range s0 - s0:c0.sentinel;
user staff_u roles { set of user's roles } \
\ level s0 range s0 - s0:c0.sentinel;
user sysadm_u roles { set of user's roles } \
\ level s0 range s0 - s0:c0.sentinel;
user system_u roles { set of user's roles } \
\ level s0 range s0 - s0:c0.sentinel;
user unconfined_u roles { set of user's roles } \
\ level s0 range s0 - s0:c0.sentinel;
user user_u roles { set of user's roles } \
\ level s0:c0 range s0 - s0:c0.sentinel;
```

As the last thing, we have to adjust the *Security context specification*. For this purpose, we can use a new security context from file *./public-MCS-4.0/policy-src/SID*. We only replace an old security context with a new one.

Finally, we will need only compile our new policy. Now, we use our compiler *mcheck-policy*. We can use the following commands:

```
cd /etc/selinux/user1/
mcheckpolicy -M -o ./policy/policy.24 ./src/policy.conf
```

Note: If you did not rename the new compiler, you will have to use the original compiler's name *checkpolicy*.

5.3.2 SELinux Policy with MLS

In the second case, we need to adjust the SELinux MLS context in current policy. Consider we have a *loadable module* policy. The *loadable module* infrastructure allows policy to be managed on a modular basis, in that there is a **base policy module** that contains all the core components of the policy (i.e. the policy that should always be present), and zero or more modules that can be loaded and unloaded as required (for example if there is a module to enforce policy for ftp, but ftp is not used, then that module could be unloaded).

As an example we use the SELinux Reference Policy. For detailed information see section 1.2. The SELinux Reference Policy can be built into a monolithic policy or a loadable module policy.

The **first** step in the adjustment of SELinux policy with MLS is the addition of our special categories to file *./refpolicy/policy/mls*. These categories can be extracted from the file *./public-MCS-4.0/policy-src/MLS declaration* and they will be placed before a call of macro *gen_cats(mls_num_cats)*.

The **second** step is the adjustment of the security levels definition. The appropriate security levels would look like this one:

```
level sx:c0.last_category;
```

Where *sx* is a sensitivity level used in policy and *last_category* is the last defined category in policy.

The **third** step is the addition of the our MLS constraints definition. These MLS constraints can be extracted from the file *./public-MCS-4.0/policy-src/MLS declaration* too.

The **fourth** step is the adjustment of the *User declarations*. We have to adjust the mls context for users that allow each user to work with our categories.

The **fifth** step is the adjustment of the *Security context specification*. We have to add an appropriate categories to the security context specified in this part. For this purpose, we can use as an inspiration security context in the file *./public-MCS-4.0/policy-src/SID*.

The **final** step is the compiling of the policy, in this case the compiling of SELinux Reference Policy. We can use simple commands:

```
cd ./refpolicy
make install
```

For detailed information see a file called *README* located in *./refpolicy/*.

5.3.3 Modified Refpolicy

Refpolicy consists of a base module and some modules. After reading a previous subsection you may think that an adjustment of refpolicy can be easily done. The modification of the whole refpolicy is a bit complex. So we prepared the modified base module.

Note 1: We use the categories c0, ...c7 as the specialized categories for the confidentiality in our solution. Unfortunately, refpolicy uses the categories c0, ..., c7 too. Thus, we have renamed our specialized categories for the confidentiality. Now we have the categories d0, ...d7 for the confidentiality. The name for the categories of confidentiality is derived from the Slovak noun *dôvernost'* "confidentiality". The remaining categories remain the same as before.

1. Extract the archive *refpolicy-2DLSMwPTS.zip*
2. Install and load our pre-prepared refpolicy

```
cd ~/refpolicy-2DLSMwPTS/src/policy
make install
make load
```

3. Compile our modified base module for refpolicy-2DLSMwPTS

```
cd ./public-MCS-4.0/policy-src/refpolicy/refpolicy-modif
checkmodule -M base.conf -o base.mod
```

4. Create the modified base module package for refpolicy-2DLSMwPTS

```
/usr/bin/semodule_package -o base.pp -m base.mod \
 -f base.fc -u users_extra -s seusers
```

5. Install the new base.pp policy package for refpolicy-2DLSMwPTS.

```
install -m 0644 base.pp /usr/share/selinux/ \
refpolicy-2DLSMwPTS
```

Note 2: We did not include the X-server classes into mlscontraint rules in base module because the security cover of X-server is still experimental in refpolicy.
Note 3: Refpolicy does not allow to set categories for user *user*. The user *user* has only sensitivity *s0* in its security context. Thus, we can not use the identity *user* in our modified refpolicy.

5.3.4 SELinux Security Context

Finally, if all is done, we **need to assign** an appropriate security context to all entities in our system, i.e. we need to determine the security context of objects and subjects. That means that we have to analyze all objects and subjects, determine the values of attributes for confidentiality and integrity, determine which object will have label, and determine which subject will be the partially trusted subject.

We can use the *type_transition* and the *range_transition* rules to assign the appropriate security context for the subject. Detailed information can be found in the section 2.4 and the subsection 2.1.5, respectively.

This may be done using the command *runcon* too. We can use the *runcon* command to run a command in a specific context. This is useful for scripting or for testing policy. The command *runcon* allows for the launching of a process into an explicitly specified context (user, role, domain, and level range), but SELinux may deny the transition if it is not approved by the policy configuration.

The following example shows how to run bash as a partially trusted subject with attributes $CW_S = 4, CLW_S = 2, IW_S = 2, ILW_S = 3$, and $CR_S = 1, CLR_S = 3, IR_S = 1, ILR_S = 0$, and labels: $CLWS_S = 3, CLRS_S = 3, ILWS_S = 3, ILRS_S = 3$.
Example:

```
runcon -l s0:c4.c_max,clS2.clS_max,i0.i2,ilS0.ilS3,cl3,il3,ps- \
s0:c0.c1,clS0.clS3,i1.i_max,ilS0.ilS_max,cl3,il3 bash
```

It is difficult to set a security context for command *runcon*. Thus, we provide a utility *launcher*. This utility is designed to help us to set the security context for command *runcon*.

Conclusions

In this thesis, we have improved our previous implementation of Two-Dimensional Labelled Security Model with Partially Trusted Subjects. We described a full evolution of our solution. We mentioned also advantages and disadvantages of our partial solutions and we compared them with our solution presented in bachelor's thesis. We made some important changes in the SELinux *kernel*, SELinux *compiler*, and *sepol* library. These changes in the SELinux *kernel* allow us to use only the categories in our final solution MCS-4.0. And on the basis of this fact we can combine our final solution MCS-4.0 with an arbitrary SELinux policy.

Although, according to the assignment, we were expected to create necessary supporting tools, it was not necessary because the standard SELinux tools suffice.

Mgr. Miroslav Hoták [15] has shown in his master's thesis that the implementation of our model by creating the prototype model using Linux Security Modules architecture is feasible. Unlike that solution, we can combine our model with an arbitrary SELinux policy.

From a more general point of view, another challenging task is to analyze all modules that are used in refpolicy. After this analysis, we will be able to set an appropriate security context for each entity of operating system from a viewpoint of our model.

Bibliography

[1] MAYER, F. - MACMILLAN, K. - CAPLAN, D.: SELinux by Example: Using Security Enhanced Linux, Prentice Hall, 2006. ISBN-10: 0-131-96369-4

[2] The official Security Enhanced Linux (SELinux) project page.
Accessed at
`http://selinuxproject.org`
[24.2.2012]

[3] Bell, D.E., La Padula L.J.: Secure Computer Systems: Mathematical Foundations and Model, Technical Report (1973)

[4] Bell, D.E., La Padula L.J.: Secure Computer System: Unified Exposition and Multics Interpretation. Technical report (1976)

[5] DoD, Trusted Computer System Evaluation Criteria, Department of Defense Standard 5200.28-STD, December 1985,
Accessed at
`http://www.radium.ncsc.mil/tpep/library/rainbow/5200.28-STD.html`
[24.2.2012]

[6] Common Criteria Project, Common Criteria Version 3.1,
Accessed at
`http://www.commoncriteriaportal.org/cc/`
[24.2.2012]

[7] Tipton, H.F., Krause, M. (editors): Information Security Management Handbook, 5th edition, CRC Press LLC (2004)

[8] Hanson, C.: SELinux and MLS: Putting the Pieces Together, Security Enhanced Linux Symposium (2006)
Accessed at
`http://selinuxsymposium.org/2006/papers/03-SELinux-and-MLS.pdf`
[24.2.2012]

[9] JANÁČEK, J.: General Purpose Operating System for Security-Critical Applications : PhD. thesis. Bratislava : Univerzita Komenského, 2010

[10] JANÁČEK, J.: A Security Model for an Operating System for Security-Critical Applications in Small Office and Home Environment. In: *Communications : Scientific Letters of the University of Žilina*. 2009, vol. 11, no. 3, pp. 5–10.

[11] JANÁČEK, J.: Mandatory Access Control for Small Office and Home Environment. In: *Informačné Technológie – Aplikácie a Teória : Zborník príspevkov prezentovaných na pracovnom seminári ITAT*. 2009, pp. 27–34.

[12] JANÁČEK, J.: Two Dimensional Labelled Security Model with Partially Trusted Subjects and Its Enforcement Using SELinux DTE Mechanism. In: *Networked Digital Technologies : Communications in Computer and Information Science 87*. Springer, 2010, pp. 259–272.

[13] JURČÍK, M.: Using SELinux to Enforce Two-Dimensional Labelled Security Model with Partially Trusted Subjects : bakalárska práca. Bratislava : Univerzita Komenského, 2010.

[14] JANÁČEK, J., JURČÍK, M.: Implemantation of two-dimesional security model with partially trusted subjects using SELinux DTE and MLS In: *Informačné Technológie – Aplikácie a Teória : Zborník príspevkov prezentovaných na pracovnom seminári ITAT*. 2011, pp. 79-84

[15] HOTÁK, M.: Implementation of Two-dimensional Labelled Security Model with Partially Trusted Subjects in Linux : diplomová práca. Bratislava : Univerzita Komenského, 2011.

[16] SMALLEY, S.: Configuring the SELinux Policy - Last revised: Feb 2005,
Accessed at
`http://www.nsa.gov/research/_files/publications/selinux_configuring_policy.pdf`
[24.2.2012]

[17] MORRIS, J.: New secmark-based network controls for SELinux
Accessed at
`http://blog.namei.org/2006/05/23/new-secmark-based-\network-controls-for-selinux/`
[24.2.2012]

[18] MORRIS, J.: A Brief Introduction to Multi-Category Security (MCS)
Accessed at
`http://james-morris.livejournal.com/5583.html`
[24.2.2012]

[19] MORRIS, J.: An Overview of Multilevel Security and LSPP under Linux
Accessed at
`http://james-morris.livejournal.com/5020.html`
[24.2.2012]

[20] BRINDLE, J.: Secure Networking with SELinux
Accessed at
`http://securityblog.org/brindle/2007/05/28/secure-networking-with-selinux/`
[24.2.2012]

[21] Wikipedia Security-Enhanced Linux
Accessed at
`http://en.wikipedia.org/wiki/Security-Enhanced_Linux`
[24.2.2012]

[22] Red Hat Selinux Guide
Accessed at
`http://www.centos.org/docs/4/html/rhel-selg-en-4/index.html`
[24.2.2012]

[23] NSA Security-Enhanced Linux
Accessed at
`http://www.nsa.gov/research/selinux/index.shtml`
[24.2.2012]

[24] Getting Started with Multi-Category Security (MCS)
Accessed at
`http://www.centos.org/docs/5/html/Deployment_Guide-en-US/sec-mcs-getstarted.html`
[24.2.2012]

[25] Building a custom kernel
Accessed at
`http://fedoraproject.org/wiki/Building_a_custom_kernel`
[24.2.2012]

[26] SELinux Reference Policy
Accessed at
`http://oss.tresys.com/projects/refpolicy`
[24.2.2012]

Appendix A

This appendix provides a list of the CD-ROM contents. The CD-ROM enclosed with this thesis contains our final implementation and previous implementations, namely MLS-1.0, MLS-2.0, and MLS-3.0. The CD-ROM also contains a complete copy of the thesis in Adobe Acrobat format, i.e. PDF.

The following directories can be found on the attached CD-ROM:

.:

launcher	- The utility launcher.
original files	- The original files of the kernel, checkpolicy, libsepol, and setools.
public-MCS-4.0	- The final solution.
public-MLS-1.0	- The partial solution, which preceded the final solution.
public-MLS-2.0	- The partial solution, which preceded the final solution.
public-MLS-3.0	- The partial solution, which preceded the final solution.
thesis	- The complete copy of the thesis in Adobe Acrobat format.

Resumé

V tejto práci sme sa snažili zlepšiť implementáciu "Two-Dimensional Labelled Security Model with Partially Trusted Subjects" využitím viacúrovňovej bezpečnosti ako jedného z prostriedkov, ktorý nám SELinux ponúka. Autorom modelu je RNDr. Jaroslav Janáček, PhD., ktorý daný model navrhol v rámci svojej dizertačnej práce [9].

Cieľom tejto práce bolo implementovať bezpečnostný model s názvom Two-dimensional labelled security model with partially trusted subjects použitím SELinux mechanizmu v operačnom systéme Linux. Od tejto implementácie sme očakávali zlepšenie implementácie prezentovanej v bakalárskej práci [13] využitím viacúrovňových bezpečnostných prvkov SELinuxu. Implementácia mala pozostávať zo SELinux politiky a podporných nástrojov pre administrátora systému a/alebo užívateľov, nakoľko sme očakávali, že takéto podporné nástroje budú potrebné.

V práci uvádzame podrobnú evolúciu nášho snaženia sa o dosiahnutie cieľov tejto práce. Popisujeme aj rôzne prístupy ako riešiť problémy, na ktoré sme narazili. Nakoľko je možné očakávať pridanie novej funkcionality pre SELinux, tak v prílohe tejto práce uvádzame aj čiastočné riešenia, ktoré predchádzali finálnemu riešeniu. Pri každom riešení popisujeme aj jeho výhody voči pôvodnému riešeniu.

Podarilo sa nám nájsť takú implementáciu tohto modelu, ktorú je možné ľahko kombinovať s ľubovolnou, už existujúcou, SELinux politikou bez potreby špeciálnych podporných nástrojov.

Výsledkom práce je implementácia, ktorá je kombinovateľná s ľubovolnou SELinux politikou len na základe využitia viacúrovňovej bezpečnosti a to bez nutnosti drastických zásahov do pôvodnej, hostiteľskej politiky.

Kľúčové slová: politika toku informácií, bezpečnostný model, SELinux politika, MLS SELinux, MCS SELinux