Univerzita Komenského

Fakulta Matematiky, Fyziky a Informatiky

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

2010 **Martin Jurčík**

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

BACHELOR'S THESIS

Martin Jurčík

Comenius University in Bratislava

Faculty Of Mathematics, Physics And Informatics
Department Of Computer Science

9.2.1 Informatics

Advisor: RNDr. Jaroslav Janáček

Bratislava 2010

Abstract

In this thesis, we try to implement a prototype model of Two-Dimensional Labelled Security Model with Partially Trusted Subjects based on PhD thesis[JJ], written by RNDr. Jaroslav Janáček. We try to show the feasibility of using SELinux to enforce this model. The prototype model, respectively prototype model policy, uses a NSA Security-Enhanced Linux (SELinux) like an interpreter.

Keywords: information flow policy, security model, SELinux policy

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, 1. June 2010 Signature: .....

Acknowledgements

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

Special thanks to my family for all their support.

Contents

Introduction

SELinux is an implementation of a mandatory access control architecture called Flask in the Linux kernel. SELinux can enforce an administratively-defined security policy over all processes and objects in the system, basing decisions on labels containing a variety of security-relevant information. The architecture provides flexibility by cleanly separating the policy decision-making logic from the policy enforcement logic.[NSA]

Chapter 1

Security models

In this chapter, we provide an overview of security models used in Linux distribution.

1.1 Standard Linux Security Model

Operating systems have two forms of access control:

- discretionary access control(DAC)
- mandatory access control(MAC)

Standard Linux security is a form of DAC. SELinux adds a flexible, configurable MAC to Linux.
Standard Linux file permissions use the Discretionary Access Control (DAC) model. Under DAC, files are owned by a user and that user has full control over them, including the ability to grant access permissions to other users. The root account has full control over every file on the entire system.

1.2 SELinux Security Model

Current Linux kernel includes SELinux security module that implements a very flexible security mechanism. SELinux provides a mechanism for supporting access control security policies through the use of Linux Security Modules (LSM) in the Linux kernel. DAC has a fundamental weakness in that, it is subject to a variety of malicious software attacks. MAC is a way to avoid these weaknesses. Most MAC features implemented so far are a form of multilevel security modeled after governmental classification controls.
SELinux implements a more flexible form of MAC called type enforcement and an optional form of multilevel security.
SELinux access control is based on a security context associated with all system resources including processes. The security context contains three elements: **user**, **role**, and **type identifiers**. The type identifier is the primary part of the access control.
Type enforcing(TE) is the primary access control feature. Access is granted between subjects(domain) and objects(object) by specifying *allow* rules that have the subject's type as the source and the object's type as the target.
SELinux provides an optional Multi Level Security(MLS) and Multi Level Category(MLC) access control mechanism that provides further access restrictions for a certain class of data sensitivity applications. We used [NSA] and [MMC] to study SELinux policy.

1.3 Two-Dimensional Labelled Security Model with Partially Trusted Subjects

In this section, we present Two-Dimensional Labelled Security Model with Partially Trusted Subjects. All about this model you can find in [JJ].

This model contains two types of entities - subjects and objects. The subjects are active entities of the model - they can perform operations on objects. Typical subjects are processes. As opposed to those the Objects are passive entities - some examples are file, directories, pipes, socket, etc.
Each entity in this model has several security attributes associated with them. 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.

Each object is assigned a confidentiality level, an integrity level and an identifier of a user that is the object's owner. The idea of this 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.
The subjects in this model are divided into three categories:
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.
A partially trusted subject is

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

The most important difference between trusted and partially trusted subjects is in the level of trust.

Chapter 2

Integration into SELinux policy

In this chapter, we provide an overview of integration *Two-Dimensional Labelled Security Model with Partially Trusted Subjects* into SELinux policy.

2.1 The Information Flow policy

In this section, we present *The Formal definition of the information flow policy* of Two-Dimensional Labelled Security Model with Partially Trusted Subjects security model.[1]

2.1.1 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),

[1] This section is a part of the [JJ]

- $U_O \in U$ – 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

$$\begin{aligned}\mathbf{read}(S,O) \overset{\text{def}}{\iff} &[CR_S \geq C_O \vee (CRL_S \geq C_O \wedge L_O \in CRLS_S) \\ &\vee (C_{appr} \geq C_O \wedge \mathbf{UserApprovedRead}(S,O))] && \text{(2.1a)}\\ &\wedge [IR_S \leq I_O \vee (IRL_S \leq I_O \wedge L_O \in IRLS_S)] && \text{(2.1b)}\\ &\wedge [U_S = U_O \vee C_O \leq C_{shareable}] && \text{(2.1c)}\\ &\wedge [U_S = U_O \vee U_O \in IRUS_S \vee IR_S \leq I_{shareable}] && \text{(2.1d)}\end{aligned}$$

   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

$$\begin{aligned}\mathbf{write}(S,O) \overset{\text{def}}{\iff} &[CW_S \leq C_O \vee (CWL_S \leq C_O \wedge L_O \in CWLS_S)] && \text{(2.2a)}\\ &\wedge [IW_S \geq I_O \vee (IWL_S \geq I_O \wedge L_O \in IWLS_S)] && \text{(2.2b)}\\ &\wedge [U_S = U_O \vee I_O \leq I_{shareable}] && \text{(2.2c)}\\ &\wedge [U_S = U_O \vee U_O \in CWUS_S \vee CW_S \leq C_{shareable})] && \text{(2.2d)}\end{aligned}$$

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

$$\mathbf{create}(S,P) \overset{\text{def}}{\iff} \mathbf{read}(S,P) \tag{2.3a}$$
$$\wedge \mathbf{write}(S,P) \tag{2.3b}$$

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

$$C_O := \begin{cases} CWL_S & \text{if } L_P \in CWLS_S \\ CN_S & \text{otherwise} \end{cases} \tag{2.3c}$$
$$I_O := \begin{cases} IWL_S & \text{if } L_P \in IWLS_S \\ IN_S & \text{otherwise} \end{cases} \tag{2.3d}$$
$$L_O := LN_S \tag{2.3e}$$
$$U_O := U_S \tag{2.3f}$$

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

$$\mathbf{delete}(S,O,P) \overset{\text{def}}{\iff} \mathbf{read}(S,P) \tag{2.4a}$$
$$\wedge \mathbf{write}(S,P) \tag{2.4b}$$
$$\wedge \mathbf{write}(S,O) \tag{2.4c}$$

5. Each untrusted subject S must satisfy:

$$CW_S = CWL_S \geq CR_S = CRL_S \tag{2.5a}$$
$$IW_S = IWL_S \leq IR_S = IRL_S \tag{2.5b}$$
$$CWLS_S = CRLS_S = IWLS_S = IRLS_S = \emptyset \tag{2.5c}$$
$$CN_S \geq CW_S \tag{2.5d}$$
$$IN_S \leq IW_S \tag{2.5e}$$
$$LN_S = 0 \tag{2.5f}$$

6. Each partially trusted subject S must satisfy:

$$CW_S \geq CR_S \tag{2.6a}$$
$$CW_S \geq CRL_S \tag{2.6b}$$
$$CWL_S \geq CR_S \tag{2.6c}$$
$$IW_S \leq IR_S \tag{2.6d}$$
$$IW_S \leq IRL_S \tag{2.6e}$$
$$IWL_S \leq IR_S \tag{2.6f}$$
$$CN_S \geq CW_S \tag{2.6g}$$
$$IN_S \leq IW_S \tag{2.6h}$$

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:

0 – public,
1 – C-normal,
2 – C-sensitive,

and the meaning of the integrity levels:

0 – potentially malicious,
1 – I-normal,
2 – I-sensitive.

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

$$\begin{aligned}
\textbf{reclassify}(S,O,c,i) \overset{\text{def}}{\iff} & [C_O \leq CR_S \wedge C_O \geq CW_S \wedge c \geq CW_S] && \text{(2.7a)}\\
& \wedge [I_O \geq IR_S \wedge I_O \leq IW_S \wedge i \leq IW_S] && \text{(2.7b)}\\
& \wedge \textbf{CanRevoke}(O) && \text{(2.7c)}\\
& \wedge U_O = U_S && \text{(2.7d)}\\
& \wedge L_O = LN_S && \text{(2.7e)}
\end{aligned}$$

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

$$\begin{aligned}
\textbf{debug}(D,S) \overset{\text{def}}{\iff} & CR_D \geq \max\{CR_S, CW_S\} && \text{(2.8a)}\\
& \wedge CW_D \leq \min\{CR_S, CW_S\} && \text{(2.8b)}\\
& \wedge IR_D \leq \min\{IR_S, IW_S\} && \text{(2.8c)}\\
& \wedge IW_D \geq \max\{IR_S, IW_S\} && \text{(2.8d)}\\
& \wedge U_D = U_S && \text{(2.8e)}
\end{aligned}$$

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

$$\begin{aligned}
\textbf{maysignal}(S,R) \overset{\text{def}}{\iff} & CW_S \leq CR_R && \text{(2.9a)}\\
& \wedge IW_S \geq IW_R && \text{(2.9b)}\\
& \wedge U_S = U_R && \text{(2.9c)}
\end{aligned}$$

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

$$\mathbf{setCN}(S,c) \overset{\text{def}}{\iff} c \geq CW_S \tag{2.10}$$

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

$$\mathbf{setIN}(S,i) \overset{\text{def}}{\iff} i \leq IW_S \tag{2.11}$$

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

$$\mathbf{setCR}(S,c) \overset{\text{def}}{\iff} c \leq CR_S \tag{2.12}$$

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

$$\mathbf{setCW}(S,c) \overset{\text{def}}{\iff} c \geq CW_S \tag{2.13}$$

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

$$\mathbf{setIR}(S,i) \overset{\text{def}}{\iff} i \geq IR_S \tag{2.14}$$

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

$$\mathbf{setIW}(S,i) \overset{\text{def}}{\iff} i \leq IW_S \tag{2.15}$$

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

$$\mathbf{setCRL}(S,c) \overset{\text{def}}{\iff} c \leq CR_S \tag{2.16}$$

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

$$\mathbf{setCWL}(S,c) \overset{\text{def}}{\iff} c \geq CW_S \tag{2.17}$$

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

$$\textbf{setIRL}(S,i) \overset{\text{def}}{\iff} i \geq IR_S \tag{2.18}$$

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

$$\textbf{setIWL}(S,i) \overset{\text{def}}{\iff} i \leq IW_S \tag{2.19}$$

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

2.2 SELinux abilities

In this section, we present a SELinux abilities, which we use to integrate our model into SELinux policy. We use UML diagrams for better understanding all dependencies.
From the following UML diagrams you can get an image about all necessary components we used in the implementation process in Chapter **3 Implementation**.

Figure 2.1 shows the attributes and types mapping from SELinux into the Two-Dimensional Labelled Security Model with Partially Trusted Subjects.
Figure 2.2 shows the syntax of user identity declarations and role declaration. Here you can see, which type category is used in the role declaration.
Figure 2.3 shows the declaration of TE Access Vector rules, concrete Allow Access Vector rule. Here you can see all SELinux classes that have many permissions. These permissions are used in **Decision Making Power Control Mechanism** in Section 3.3 .
Figure 2.4 shows constraints definition and dependence rules defined in previous Section(2.1). The Constraints are the base elements of Control Mechanism.
Last Figure 2.5 shows the rules, which we can use to restrict the ability to set the arbitrary type to an actually created object.

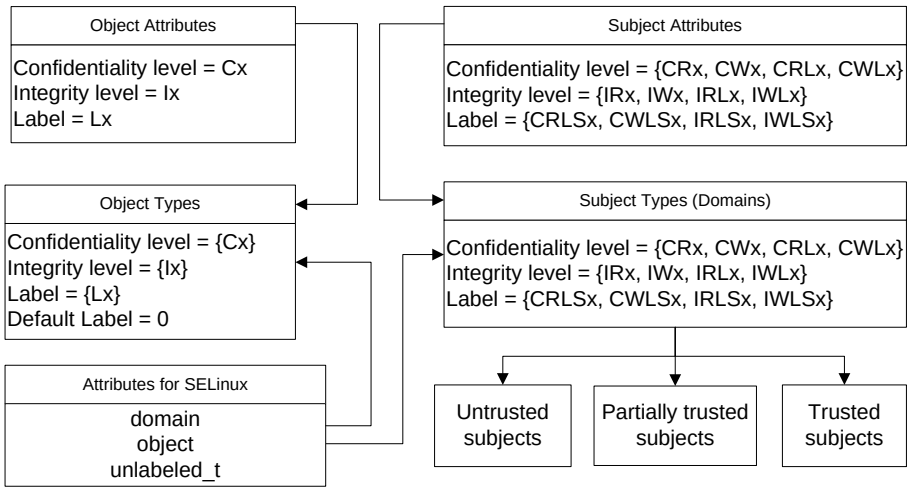

Figure 2.1: Types & Attributes definition

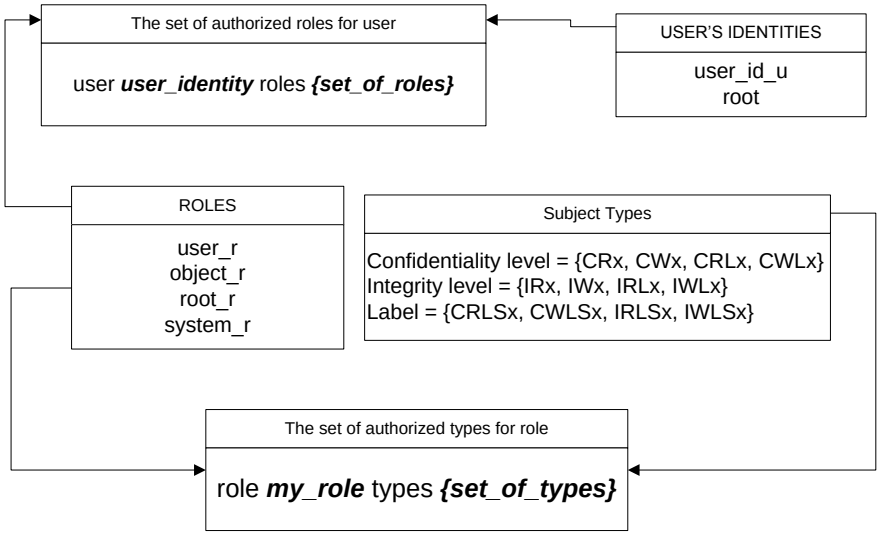

Figure 2.2: Role and User Identity declarations

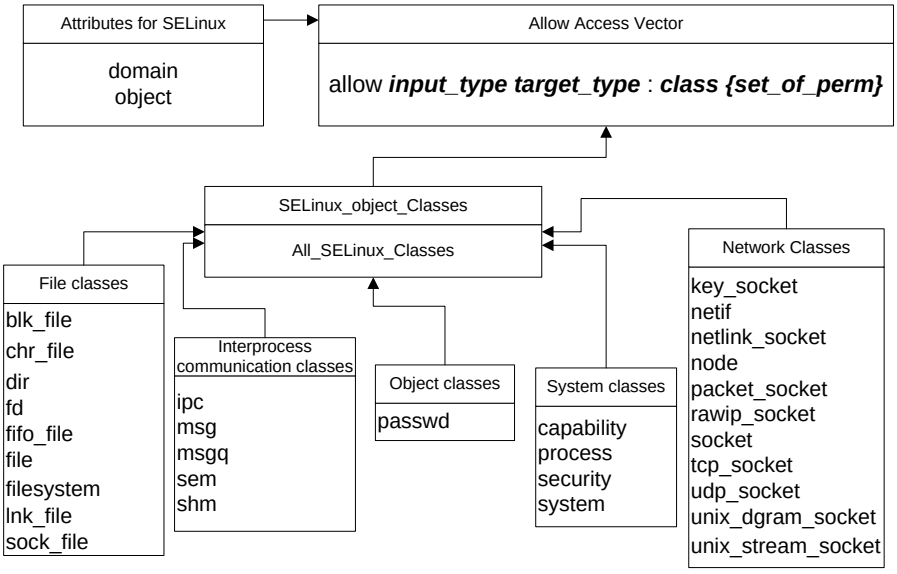

Figure 2.3: Allow Access Vector definition

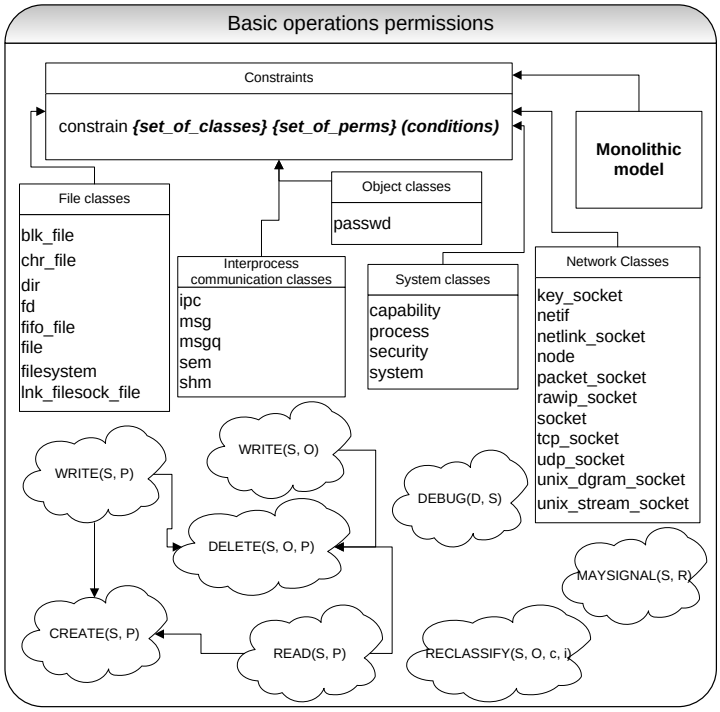

Figure 2.4: Constraints definition

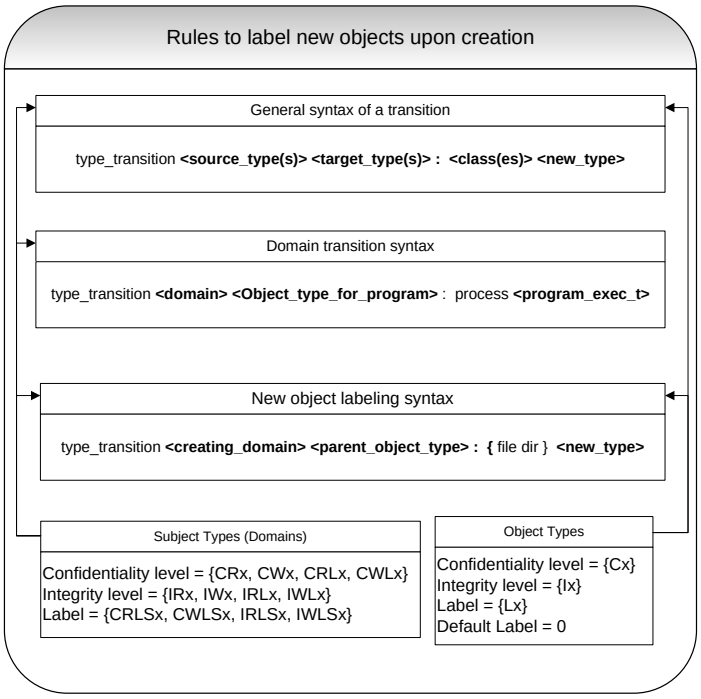

Figure 2.5: Definition of new type for created object

Chapter 3

Implementation

In this chapter, we provide an overview of our implementation process.

3.1 Prepare

We decided to use a virtual machine in VMware Player for quick ”back-up and restore” agility. In process of writing this policy we killed ca. 20 or 25 virtual machines.
We decided to use a Fedora12 virtual machine from [Virtual Machine], but in original version was is unable to log as root. We had to edit the core settings to enable to log as root.
This distribution contains an older version of SELinux. We had to update to the latest version for kernel, specifically 2.6.31.5-127.fc12.i686.

```
[root@root ~]# yum update libselinux selinux-policy-targeted
```

and

```
[root@root ~]# yum install selinux-policy-devel \
selinux-policy-strict policycoreutils-newrole selinux-policy-mls
```

But all of these policy versions (strict, targeted or mls) don’t contain the policy source files, only the compilated version.
Then we decided to use SELinux Reference Policy[refpolicy]. The policy source file ’policy.conf’ has cca $2,9 * 10^6$ lines of code and it uses lots of macros.

3.2 Making of SELinux Policy or 'Busy Days'

The policy for SELinux is a binary representation that can be loaded into the kernel. A program **checkpolicy** is a compiler for SELinux policy. A source for checkpolicy is policy.conf. Each policy for SELinux must consist 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

The TE model provides fine-grained control over processes and objects in the system and the RBAC model provides a higher level of abstraction to simplify user management.

3.2.1 Writing the policy - Yes, and which way??

We took the policy.conf file from the refpolicy. Then we had to decide which way we should go. We had three options:

1. To write a minimalistic monolithic policy,
2. To use the refpolicy and adapt it for our model,
3. To use MLS and MLC in refpolicy and adapt them for our model

It would be difficult to modify the refpolicy. Therefore we decided to apply the first option. Using two other options would require to edit all $2,9 * 10^6$ lines of code.

3.2.2 FLASK definitions

In this subsection, we present all components that we use for configuring of SELinux. All what we need to implement is in the Part **2.1.1**.

First of all, we must gain a FLASK definitions. We extract it from the refpolicy policy.conf file. The FLASK definitions consist of the security object classes, initial security identifiers, and common prefixes for access vectors. In our case, the FLASK definitions from the refpolicy was not complete. We must add *module_request* to the class "system", and *acceptfrom*, *newconn* and, *connectto* to the class "dccp_socket". Now, the Flask classes are compatible with our current Fedora12 kernel 2.6.31.5-127.fc12.i686.

3.2.3 TE Statement

In this part of policy.conf we declare attributes and types for our policy.

A type attribute is a name that can be used to identify a set of types with a similar property. Each type can have any number of attributes, and each attribute can be associated with any number of types.
We use specialty attribute *object* for Object types and specialty attribute *domain* for Subject types.

Objects

Our model uses four security attributes for objects - the confidentiality and integrity levels, the owner user identifier, and the label used to describe the sets of designated inputs and outputs for partially trusted subjects. We use SELinux user field in object labels to represent object owners. Firstly, we define a type attribute for each confidentiality level (specifically: C0, C1 and C2). Then, we define a type attribute for each integrity level(specifically: I0, I1 and I2) and finally we define a type attribute for each used label (Lx for label x, we have actually only one Label - default label - L0).

Example:

```
##type attribute for confidentiality level
attribute C0;
attribute C1;
attribute C2;
```

Now, if we have a complete type attribute, we can define a types for object. We define a type for every possible pair of confidentiality and integrity levels, and tag it with two type attributes as follows:
For every *c,i* $\in \{0,1,2\}$ add a type tagged with the attributes *Cc* and *Ii*. Then, we add new object types for every combination of the confidentiality, integrity levels and labels and tag them with the three type attributes - *Cc, Ii, Ll* where *c* and *i* are the same as above and *l* is its label.

Example:

```
##OBJECT##
#confidentiality x integrity
type Obj_C1I1, object, C1, I1;

#confidentiality x integrity x Label
% #standard case for default label
type Obj_L0C1I1, object, C1, I1, L0;
```

The Partially Trusted Subject (see below) has access to the Obj_L0C1I1, only if has the required attributes, but not to the Obj_C1I1.

Subjects

Firstly, we define a type attribute for each confidentiality level for subjects (specifically: CR*x*, CW*x*, CRL*x*, CWL*x*) for $x \in \{0,1,2\}$. Then, we define a type attribute for each integrity level for subjects (specifically: IR*x*, IW*x*, IRL*x*, IWL*x*) for $x \in \{0,1,2\}$, and finally we define a type attributes CRLS*x*, CWLS*x*, IRLS*x* and IWLS*x* for $x \in L$, where L is the set of needed labels.

Example:

```
##ATTRIBUTES FOR SUBJECTS
#CRx
attribute CR0;
#CWx
attribute CW0;
#CRLx
attribute CRL0;
#CWLx
attribute CWL0;
```

Our model distinguishes trusted, untrusted, and partially trusted subjects. The full set of needed types for trusted and untrusted subjects contains 81 types. These types are to be tagged with the attributes $CR\langle CR_S\rangle, CW\langle CW_S\rangle, IR\langle IR_S\rangle$ and $CR\langle CR_S\rangle$, where $\langle x\rangle$ denotes the value of x.

We add a new type with a unique name for every used combination of the partially trusted subject attributes, and tag it with the following type attributes: $CR\langle CR_S\rangle$, $CW\langle CW_S\rangle$, $IR\langle IR_S\rangle$ and $CR\langle CR_S\rangle$, where $\langle x\rangle$ denotes the value of x, and $CRLSy$ for $y \in \mathbf{CRLS}_S$, $CWLSy$ for $y \in \mathbf{CWLS}_S$, $IRLSy$ for $y \in \mathbf{IRLS}_S$ and $IWLSy$ for $y \in \mathbf{IWLS}_S$.

Example:

```
##TRUSTED SUBJECTS##
type trS_0_0_0_1, domain, CR0, CW0, IR0, IW1;
##UNTRUSTED SUBJECTS##
type unS_0_0, domain, CR0, CW0, IR0, IW0;
##PARTIALLY TRUSTED SUBJECTS##
ptS_C1102_I2110_L0000, domain, CR1, CW1, CRL0, \
CWL2, IR2, IW1, IRL1, IWL0, CRLS0, CWLS0, IRLS0, IWLS0;
```

TE Access Vector Rules

A TE access vector rule specifies a set of permissions based on the type pair and object security class. These rules can be specified for each kind of access vector, including the allowed, auditallow, and auditdeny vectors. The first rule is very important to us. The auditallow(auditdeny) vectors we can use to audit (record avc messages into audit.log file) all allowed (denied) operations.
Access control rules are specified using the following syntax:
allow *source_types target_types* : *classes permissions ;* . In our case, the source_types is a set of subjects(attribute domain) and the target_types is a set of objects(attribute object), and subjects(attribute domain). The *permissions* represent a set of operations. The *classes* is a set of classes that the rule applies to. Classes are used in SELinux to distinguish between different sorts of objects, such as directories, file, fifo_file, socket, process, etc.
Example:

```
allow {domain} {domain object}:dir { getattr relabelto \
unlink ioctl execute append read setattr swapon write \
lock create rename mounton quotaon relabelfrom link \
search rmdir remove_name reparent add_name open };
```

3.2.4 RBAC Statement

We use a role declaration from the refpolicy. A role declaration specifies the name of the role and the set of subject types for which the role is authorized. We don't use all abilities of RBAC, we only declare the basic set of roles. In other words, all roles in our policy are authorized to use all subject types(domains). In policy we have these roles: *system_r* for all system processes, *object_r* for all system files, *user_r* for all users, and *staff_r* specially for root. We have declared *sysadm_r* and *unconfined_r* too, but now they are not used.
Example:

```
role system_r types { abrtd_exec_t avahi-daemon_exec_t \
browser_exec_t canberra-gtk-play_exec_t dbus-daemon_exec_t \
evolution_exec_t gnome-settings-daemon_exec_t \
Launcher_exec_t nautilus_exec_t xorg_exec_t Loading_boot_t \
NetworkManager_exec_t ps_exec_t unS_0_0 unS_0_1 unS_0_2 \
unS_1_0 unS_1_1 unS_1_2 unS_2_0 unS_2_1 unS_2_2 };
```

3.2.5 User Declarations

The user declarations define each user recognized by the policy and specify the set of authorized roles for each of these users. In policy we have these users: *root* is user identity for root, *system_u* is defined for system processes and objects, and *user_**id**_u* is defined for every unprivileged users in the system, where **id** is an **user id** in system. In our case, it would be *user_500_u*. We didn't use only *user_u* identity for all users in system with more then one user, because this user field we use to represent owner.
Example:

```
user root roles { object_r staff_r system_r }
user user_u roles { object_r user_r };
user test_u roles { object_r user_r };
```

An user identity *user_u* is defined, because in our system we have only one real user(without root). An user identity *test_u* is defined only for testing the functionality.

Each SELinux policy(strict, targeted, mls, or our policy) contains in its root directory a special file - *seusers*, that is used to mapping system user onto SELinux user identity and role. For testing the functionality we can use command 'id -Z'.

Example:

```
[root@root ~]# id -Z
root:staff_r:Loading_boot_t
```

Our security context is: $user-root; role-staff\_r; type(domain)-Loading\_boot\_t$

3.2.6 Constraint Definitions

The constraint definitions specify additional constraints on permissions in the form of boolean expressions that must be satisfied. The set boolean expressions based on the user identity, role, or type attributes in the pair of security contexts. The same constraint can be imposed on multiple classes and permissions. Now, we can create the SELinux constraints corresponding to the rules of our model's information flow policy(Section 2.1). More about concrete constraints you can find in the next Section 3.3

3.2.7 Security Context Specifications

The security contexts specifications provide security context for various entities such as initial SIDs, filesystem entries, and network objects.

Initial SID Contexts

Initial SIDs are SID value that are reserved for system initialization or predefined objects. The initial SID contexts configuration specifies a security context for each initial SID. This security context consists of a user identity, a role and a type.
Here is an example:

```
sid port system_u:object_r:Obj_L0C1I1
sid devnull system_u:object_r:Obj_L0C1I1
sid kernel system_u:system_r:Loading_boot_t
sid unlabeled system_u:object_r:unlabeled_t
```

Filesystem Labeling Behaviors

The labeling behavior for a filesystem type can be specified using the *fs_use* or using the *genfs_contexts* configuration. For disk-based filesystem types that support extended attributes and the security xattr namespace, we use *fs_use_xattr*. For pseudo filesystem types representing pipe and socket objects, we use *fs_use_task*. For pseudo filesystem representing pseudo terminals and shared memory or temporary objects we use *fs_use_trans*.
Here is an example:

```
fs_use_xattr ext3 system_u:object_r:Obj_C1I1;
fs_use_task pipefs system_u:object_r:Obj_C1I1;
fs_use_trans tmpfs system_u:object_r:Obj_C1I1;
```

Genfs Contexts

The *genfs_contexts* configuration is consulted to determine a security context based on the filesystem type, the file pathname, and optionally the file type.
Here is an example:

```
genfscon ntfs / system_u:object_r:Obj_L0C1I1
genfscon proc /net system_u:object_r:Obj_L0C1I1
genfscon selinuxfs / system_u:object_r:Obj_L0C1I1
```

3.3 Control Mechanism

The Task of Control Mechanism is to restrict the information flow using the SELinux constraints corresponding to the rules of our model's information flow policy defined in Section 2.1.

The SELinux Constraint

The constraint statement has three elements: a set of object classes to which the constraint applies, a set of permissions for those classes that are being constrained, and a Boolean expression of the constraint. The constraint statement enables us to restrict specified permissions for specified object classes by defining constraints based on relationships between source and target security contexts.

```
constrain class_set perm_set expression;
```

All permissions must be valid for all object classes in the *class_set*. An *expression* is a Boolean expression of the constraint. The Boolean expression syntax supports the following keywords:

- *t1, r1, u1* - Source type, role and user, respectively
- *t2, r2, u2* - Target type, role and user

We use only following Constraint expression operators:

- == - Set member of or equivalent
- *!=* - Set not member of or not equivalent

In general, the constraint expressions can be very complex. But we usually use only complex constraint expressions, because 4 is the maximal depth, which the compiler allows us to use for constraint rules.

We defined the constraints for all operations like **read**(S,O), **write**(S,O), **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 permission in SELinux for **read**(S,O) and **getattr**(S,O) is *read*, for **write**(S,O) and **setattr**(S,O) is *write*, for **create**(S,O) are *create* and *link*, for **debug**(D,S) is *ptrace* and for **maysignal**(S,R) is correponding class *process* and permissions *sigkill*, *sigstop* and *signal*. For all **Changing the subject's security** attributes operations is corresponding permission class *process* and permissions *transition*.

```
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})
);
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})
);
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 3.1: SELinux constraints for **read**(S,O)

3.3.1 Problematic operations

A subject S may **delete** an object O from (or related to) an object P if **delete**(S,O,P) is true. In one constraint we don't have expression keywords for Source type, Target types and for Parent type of Target. Fortunately, the operation **delete**(S,O,P) can be implemented as permission *unlink*.

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. For this operation we don't have to control only Source type and Target type. We have to control Source type, Old type of Target, and New type of Target. Here are constrains probably useless. But SELinux supports a special second constraint statement called **validatetrans**. This statement was added as a part of modified multilevel security features. With the *validatetrans* we can further control the ability to change the security context of *supported objects class*.

The SELinux Constraint - Validatetrans

The *validatetrans* statement restricts the ability to change the security context of specified supported objects by defining constraints-based relationships with old and new security context and the security context of the process(subject[domain]).

```
validatetrans class_set expression;
```

In contrast with constraints, the perm_set is missing here. The Boolean expression syntax supports the following keywords:

- *t1, r1, u1* - Old type, role and user, respectively
- *t2, r2, u2* - New type, role and user, respectively
- *t3, r3, u3* - Process type, role and user

We use only following Validatetrans expression operators:

- == - Set member of or equivalent
- *!=* - Set not member of or not equivalent

But the validatetrans cannot control, whether the Old type user is equivalent to Process type user. Therefore, we have to use constraints with permission *relabelto* to control this equality. Thus, the operation **reclassify**(S, O, c, i) is implemented as combination of constraint with permission *relabelto* and validatetrans constraint for supported classes.

3.3.2 Interprocess and Network communications

The Two-Dimensional Labelled Security Model with Partially Trusted Subjects contains the rules for information flow only between subjects(domains) - source and objects - target. When we are trying to implement these rules for Interprocess and Network communications into our prototype policy, we found some problems.

3.3.3 Network communication

The first problem was a method "How to implement Network communication". Firstly, we tried to use a SELinux abilities to set a Network Object Contexts, which permits the specification of security contexts for ports, network interfaces, and nodes. But in current version of SELinux we can only control the access to ports. On the other hand, in current SELinux version, we can use an **IPtables** to set a security context for each packet. You can find a basic principles in [JB] and [JM].

Solution

The Internet is a **potentially malicious** environment. It means, we use a IPtables to label each packet from the Internet with the label for public and potentially malicious object. In our case, the corresponding label is *Obj_C0I0*. If we want to have Internet access, we have to use a specific browser that is authorized to work with this object type. In our point of view, if we want to connect to the Internet banking provider(IB provider), then this communication is **trusted** and the packets that are sending to/from IB provider **aren't** public and potentially malicious. Therefore, these packets are labeled with confidentiality-normal(C1) and integrity-normal(I1) attributes. In our case, the corresponding label is *Obj_C1I1*. In prototype policy directory the directory "script" contains a file "selinux-network.sh" that contains all commands for IPtables. To control the packet flow we use the constraints again, i.e. constraint for class ***packet***, and permissions ***send***, and ***recv***. The permission *send* is defined like *write* and the permission *recv* is defined like *read* in 2.1.1.

A real example of this situation is described in Chapter "Testing", specifically Section 4.3.

3.3.4 Interprocess communication

The second problem was a method "How to implement Interprocess communication". Linux and SElinux use for interprocess communication AF_UNIX_datagram Sockets, respectively AF_UNIX_stream Sockets. These correspond to class *unix_dgram_socket*, respectively to *unix_stream_socket*. SELinux contains also classes like *socket* and *sock_file*. Which class and permissions we have to use to control interprocess communication? Neither from the source code has not been clearly about, what SELinux really doing during the interprocess communication. The method "Trial and error" was not useful. We used [BSD Sockets Interface Programmer's Guide] to study the Interprocess communication. We wrote a simple client and server application using *AF_UNIX_datagram socket* and *AF_UNIX_stream socket*. With these applications we mapped all interprocess activities.

AF_UNIX_datagram

In case of using *AF_UNIX_datagram socket*:
The server application process creates a socket(constraint class *unix_dgram_socket*). As follow, the server application performs *bind, write, add_name* operations. Then, it creates a object(constraint class *sock_file*). The created socket inherits the security context from parent(the server application). We cannot control this inheritance. Finally, the server application performs *read* on the created socket.
The client application process performs the same operations like the server application with the difference, that the last operation is not *read* but *write* on the created socket.

The interprocess communication between client and server applications is then made with calling operation *sendto*(constraint class *unix_dgram_socket*) from the client side.

AF_UNIX_stream

In case of using *AF_UNIX_stream socket*:
The server application process creates a socket(constraint class *unix_stream_socket*). As follow, the server application performs *bind, write, add_name* operations. Then, it creates an object(constraint class *sock_file*). The created socket inherits the security context from parent(the server application). Also, we cannot control this inheritance. Finally, the server application performs *listen* and *accept* on the created socket.
The client application process performs only *connectto*(constraint class *unix_stream_socket*).

Solution

As mentioned above, the Two-Dimensional Labelled Security Model with Partially Trusted Subjects doesn't contain the rules for information flow between subjects(domains) - source and subjects(domains) - target (without **debug**(D,S) and **maysignal**(S,R)). We have to therefore define the rules for the information flow between subjects and subject.
Idea: The source subject(client) must be able to write a file and then, the target subject(server) must be able to read this file.

Formal definition:
A subject S_1 may **sendto** or **connectto** to a subject S_2 if **sendto**(S_1,S_2) is true, where

$$\textbf{sendto}(S_1,S_2) \overset{\text{def}}{\Longleftrightarrow} [CW_{S_1} \leq CR_{S_2} \vee [CWL_{S_1} \leq CRL_{S_2} \wedge (CRLS_{S_2} \cap CWLS_{S_1} \neq \emptyset)] \tag{3.1a}$$

$$\wedge [IR_{S_2} \leq IW_{S_1} \vee [IRL_{S_2} \leq IWL_{S_1} \wedge (IWL_{S_1} \cap IRLS_{S_2} \neq \emptyset)] \tag{3.1b}$$

$$\wedge [(U_{S_1} = U_{S_2}) \vee (CW_{S_1} \leq 1 \wedge IW_{S_2} \leq 1)] \tag{3.1c}$$

$$\wedge [(U_{S_1} = U_{S_2}) \vee (CWUS_{S_1} \cap IRUS_{S_2} \neq \emptyset)] \tag{3.1d}$$

Implementation:
We can use this rule for definition of the constraint for class *unix_dgram_socket* and permission *sendto* and the constraint for class *unix_stream_socket* and permission *connectto*.
Example:

```
(3.1a)
constrain { unix_dgram_socket } { sendto } (
[ (T1 == CW0) or (T1 == CW1 and T2 == { CR1 CR2 })
or (T1 == CW2 and T2 == CR2) ) ]
or [
[(T1 == CWL0) or (T1 == CWL1 and T2 == { CRL1 CRL2 })
or (T1 == CWL2 and T2 == CRL2)]
and
(T1 == { CWLS0 } and T2 == { CRLS0 })
]);
```

```
(3.1b)
constrain { unix_dgram_socket } { sendto } (
[ (T1 == IW2) or (T1 == IW1 and T2 == { IR1 IR0 })
or (T1 == IW0 and T2 == IR0) ]
or [
[(T1 == IWL2) or (T1 == IWL1 and T2 == { IRL1 IRL0 })
or (T1 == IWL0 and T2 == IRL0) ]
and
(T1 == { IWLS0 } and T2 == { IRLS0 } )
] );

(3.1c and 3.1d)
constrain { unix_dgram_socket } { sendto } (
[(U1 == U2)or (T1 == { CW0 CW1 } and T2 == { IR0 IR1 })]
and
[(U1 == U2) or (U2 == { system_u })]
);
```

3.3.5 X Server

The graphical user interface in Linux is implemented using X11 protocol(X server). It controls output devices and input devices. The applications connect to the X server, send their requests to display windows, and receive events. SELinux contains the specific X Server Object Classes. With these classes, we can restrict e.g. access to the Cursor object, Drawable object, Keyboard Device Object, Screen Device Object, etc. But now, they are not used in prototype. We defined only one type, specifically *xorg_exec_t*. All subjects can work with this subject type.

Chapter 4

Testing

In this chapter, we present a tutorial for installation our policy into SELinux and some example of its using.

4.1 The installation process

This section describes the installation process.

1. Copy the contain of the archive to ”/etc/selinux/”.
2. Open ”/etc/selinux/config”.
3. Set ‘SELINUX=permissive‘ and ‘SELINUXTYPE=bakalarka‘.
4. Open ”/etc/rc.local” and add this code

```
#
# SELinux policy scripts
#
sh /etc/selinux/bakalarka/script/selinux-label.sh
sh /etc/selinux/bakalarka/script/selinux-network.sh
```

   If you don’t want to allow our model controlling your network, then remove the line containing ’selinux-network.sh’ from ”rc.local”.

5. Open the terminal and type ”touch /.autorelabel” - this command sets a new label for each entity after the reboot.

6. Then restart the computer. After that, check ”/var/log/audit/audit.log”

7. If you don’t see any ‘denied‘ AVC message, then you can open ”/etc.selinux/config” and set ‘SELINUX=enforcing‘.

4.2 Filesystem labelling

In this section, we present a filesystem labelling.
Firstly, any changes of security context(file label) are **dangerous**. Any changes we must do only in **permissive mode** - command ”setenforce 0” and in ”/etc.selinux/config” to set ‘SELINUX=permissive‘. Then, to change a label of specific file, we have to edit ”/etc/selinux/bakalarka/contexts/files/file_contexts.local”.
Example:

```
/usr/sbin/NetworkManager  system_u:system_r:NetworkManager_t
/root/sensitiveData/private_key root:staff_r:Obj_L0C2I2
```

Finally, command ”touch /.autorelabel” sets a new label for each entity after the reboot. We use this command to view all dependencies.
With this method we discovered the problem, when we tried to relabel the file ”/etc/resolv.conf”, but nothing happened. The security context of *resolv.conf* was same as before. The problem - browser, type Obj_C0I0, tried to read *resolv.conf*, but it couldn’t. Effective solution was to create a new type, not for browser, for subject - DHCP NetworkManager(NetworkManager_exec_t). Then, we defined the new type_transition rule for class file. Finally, the browser worked correctly.

All system directories should have the label Obj_C0I2. All user directories should have the label Obj_C1I1. But, the download directory for data from the Internet, should have the label Obj_C0I0.

The shared directories should have this security context *system_u:system_r:Obj_C1I1*.
The all sensitive data(files) can have the label Obj_C2I1, Obj_C1I2, or Obj_C2I2.

4.3 Utilities

In directory "..bakalarka/util/" you can find our utility called launcher, that allows you to test various security context.
Example of an unsecured browser:

1. Create a new launcher with the name 'unsecured_browser'
2. Type into command field

```
etc/selinux/bakalarka/util/launcher -x /usr/bin/firefox \
 -t browser_exec_t
```

3. Launch
4. Now, only if the network ability of our model is enabled, you can connect to the Internet, but you cannot connect to any Slovak bank portal.

In "/var/log/audit/audit.log" you will find some "denied messages" as shown below:

```
type=AVC msg=audit(1272638399.968:35): avc:  denied \
{ send } for  pid=1687 comm="firefox" saddr=192.168.26.147 \
src=57064 daddr=213.215.88.244 dest=80 netif=eth5 \
scontext=root:staff_r:browser_exec_t \
tcontext=system_u:object_r:Obj_C1I1 tclass=packet

type=AVC msg=audit(1272638400.005:36): avc:  denied \
{ recv } for  pid=1708 comm="firefox" saddr=213.215.88.244 \
src=80 daddr=192.168.26.147 dest=57064 netif=eth5 \
scontext=root:staff_r:browser_exec_t \
tcontext=system_u:object_r:Obj_C1I1 tclass=packet
```

Is it correct if we can't connect to an Internet banking? No, of course not.
And here is the solution:

Example of secured browser:

1. Create a new launcher with name 'secured_browser'
2. Type into command field

```
etc/selinux/bakalarka/util/launcher -x /usr/bin/firefox \
-t unS_1_1
```

3. Launch

4. Now, only if the network ability of our model is enabled, you can as usual connect to your favorite Slovak bank, but not to the whole Internet world.

Example of a partially trusted e-mail client:
We use the e-mail client called *Evolution*. For electronic e-mail signature, the e-mail client needs access to a private key, in our case Obj_L0C2I2, and certificates, in our case Obj_C0I2.

```
type evolution_exec_t, domain, CR0, CW2, CRL2, CWL0, IR1, \
IW1, IRL1, IWL2, CRLS0, CWLS0, IRLS0, IWLS0;
```

Conclusions

The objective of this thesis was to create a prototype SELinux policy. We have created a prototype Selinux policy and we can demonstrate the basic principles of Two-Dimensional Labelled Security Model with Partially Trusted Subjects in practice. We have shown that the using SELinux to enforce this model is feasible. We have presented several concrete applications examples used in this environment. In the future, we want to continue in our work. We want to do the fully-fledged SELinux policy without any exception in constrain rules.
We can see the following tasks that might follow this thesis:

1. X Server - to include all SELinux X Server Classes into constraint rules
2. Labelling - to assign to each entity the right label with corresponding attributes
3. Software Development Tools - to create tools set, allowing easily configure SELinux policy
4. Modular policy - to allow working with modules from targeted policy

Bibliography

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

[JJ1] JANÁČEK, J.: Two Dimensional Labelled Security Model with Partially Trusted Subjects and Its Enforcement Using SELinux DTE Mechanism. In: Communications in Computer and Information Science : Proceedings of The Second International Conference on Networked Digital Technologies. Springer, 2010 (to appear).

[JJ2] JANÁČEK, J.: Mandatory Access Control for Small Office and Home Environment. In: VOJTÁŠ, P. (ed.): Informačné Technológie – Aplikácie a Teória : Zborník príspevkov prezentovaných na pracovnom seminári ITAT. Seňa : PONT s.r.o., 2009, pp. 27-34.

[NSA] SMALLEY, S.: Configuring the SELinux Policy - Last revised: Feb 2005 http://www.nsa.gov/research/_files/publications/selinux_configuring_policy.pd f

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

[Virtual Machine] Fedora12 virtual machine: http://wwvv.quotrader.org/system/software/fedora/12/

[JM] MORRIS, J.: New secmark-based network controls for SELinux http://blog.namei.org/2006/05/23/new-secmark-based-network-controls-for-selinux/ [2.5.2010]

[JB] BRINDLE, J.: Secure Networking with SELinux http://securityblog.org/brindle/2007/05/28/secure-networking-with-selinux/ [2.5.2010]

[BSD Sockets Interface Programmer's Guide] HP 9000 Networking: BSD Sockets Interface Programmer's Guide
http://docs.hp.com/en/B2355-90136/B2355-90136.pdf
[2.5.2010]

[SELinux OC & P Reference] SELinux Object Classes and Permissions Reference
http://selinuxproject.org/page/ObjectClassesPermsg

[refpolicy] http://oss.tresys.com/projects/refpolicy
Note: 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.

Abstrakt

V tejto práci sa snažíme implementovať prototyp ”Two-Dimensional Labelled Security Model with Partially Trusted Subjects” modelu, ktorý navrhol vo svojej dizertačnej práci RNDr. Jaroslav Janáček [JJ]. Snažíme sa preukázať použiteľnosť SELinuxu pre uplatnenie tohto modelu. Prototyp, resp. prototyp politiky, využíva NSA Security-Enhanced Linux (SELinux).

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