Comenius University in Bratislava
Faculty Of Mathematics, Physics And Informatics

Implementation of Two-dimensional Labelled Security Model with Partially Trusted Subjects in Linux

Diploma thesis

67098b9e-b7d5-4a36-ac99-9314615d298f

Study Program: Informatics
Branch of Study: 9.2.1 Informatics
Department: Department of Computer Science
Advisor: RNDr. Jaroslav Janáček, PhD.
Bratislava, 2011

Bc. Miroslav Hoták

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

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

Názov : Implementation of Two-dimensional Labelled Security Model with Partially Trusted Subjects in Linux

Cieľ : The goal of this thesis is to implement our two-dimensional labelled security model with partially trusted subjects in the Linux operating system using the Linux Security Modules framework. The implementation is expected to consist of the kernel patch and the necessary user-space utilities.

Vedúci : RNDr. Jaroslav Janáček, PhD.

Dátum zadania: 18.11.2009

Dátum schválenia: 18.02.2011

.....
prof. RNDr. Branislav Rovan, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Dátum potvrdenia finálnej verzie práce, súhlas s jej odovzdaním (vrátane spôsobu sprístupnenia)

.....
vedúci práce

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

.....

I would like to thank my advisor RNDr. Jaroslav Janáček, PhD. for valuable pieces of advice and support throughout writing this thesis.

Special thanks belongs to my family for their support.

Abstract

In this thesis, we try to implement a prototype model of Two-Dimensional Labelled Security Model with Partially Trusted Subjects in Linux operating system. The security model is based on PhD thesis[1] written by RNDr. Jaroslav Janáček, PhD. We try to show the feasibility of implementation in the form of a kernel patch using Linux Security Modules architecture. The implementation also includes neccesary user space utilities for managment of the policy.

Keywords: information flow policy, security model, Linux Security Modules, POSIX capabilities

Contents

Introduction

Linux Security Modules is a security framework for Linux kernel enabling implementation of arbitrary security policies implementing mandatory access control. It enables mediating access to all security relevant kernel structures during various kernel operations triggered by system calls from user space. It also adds security fields to these structures which enable binding the policy relevant security information to the structures during their life cycle.

Additionaly, Linux Security Modules encourages the use of extended attributes in the security namespace used to store permanent security information for objects of filesystem.

Chapter 1

Access control security mechanisms

The goal of this chapter is to give an overview of various approaches to protecting resources and information using access control mechanisms and describe existing implementations of respective access control types in Linux operating system. I will not mention other security mechanisms, since properly designed operating system should mediate all security critical operations and maintain the real security contexts of concerned entities. Therefore it should be able to enforce chosen security policy by means of access control.

Access control policy in general defines rules telling whether particular subject may execute requested operation on particular object. The two most common access control principles used in operating systems are discretionary and mandatory. They may be used simultaneously.

1.1 Discretionary access control

Discretionary access control has three characteristic features:

1. it restricts access to objects on the basis of the identity of subjects, eventually it may consider group to which the subject belongs, this approach does not make difference between subjects with the same identity regardless of their other security context

2. subjects with certain access permission are able to pass this permission to other subjects

3. every object usually has the owner, it is typically derived from the subject, which created the object, access policy for the object is determined by its owner

1.1.1 DAC in Linux

Linux, similarly to all POSIX compliant systems, implements Unix permissions [2]. Operating system is a multi-user enviroment. User may correspond to a real person, or to a role executing specific system operations. Users may be aggregated into user groups. The idea behind groups is to provide the same access permissions for all users in the group.

Each existing process runs on behalf of a user. Each filesystem object is owned by one of the users and is assigned a group. Setting access policy for the object is handled by its owner. Objects keeps three distinct sets of permissions. These are the owner, the group and the others set.

When a process attempts to access an object, the user on whom behalf the process runs is compared with the owner of the object and in case of a match, the owner permission set is used for access control. Otherwise, the membership of process user in object group is tested and in case of positive answer the group permission set is used. If neither of above is true, then the other set permissions are chosen.

Each set comprises of three permission types, having slightly different semantics depending on the object type:

- *read* - in case of a file grants the right to read the file, when set on a directory enables to list the content of the directory, for example using command *ls*

- *write* - enables modification of a file, in case of a directory creating a new file or removing existing file within the given directory, in the latter case the write permission on particular file is not controlled

- *execute* - enables executing a file as a program, in case of a directory enables traversing its subtree and accessing its contents, for example to read a file, just read permission is not sufficient, all directories on path from the root to the parent directory of the file must grant execute permission

The specific role in discretionary access control is bestowed upon system administrator, who bypasses all control checks and who is authorized to change DAC permission sets on arbitrary object, regardless of being its owner.

In order to flatten major differences between the authority of normal users and the system administrator, following two permission types were added to executable files to enable granting such functionality to normal users, that would elsewhere be prohibited.

- *setuid*
- *setgid*

These permissions belong to the same category as *execute* flag, *setuid* might be in the owner set of permissions, *setgid* in the group set. Therefore a file might have either *execute* or *setuid* or no permission set in the owner set at the third position. The same is true for the *setgid* permission and the group set of permissions.

To describe the semantics of *setuid* and *setgid*, I will explain details about the process user identity and how the new process user identity is computed when a parent process executes binary file.

The user identity of a running process is kept in several types of user identity numbers, each uniquely determining the user registered in the system. The real uid number determines the identity of user who started the process. The effective uid number determines the security capabilities of a given process. During all discretionary access control checks the effective uid number of process is taken.

Under normal circumstances, the real uid and the effective uid number of running process is equal. When a process tries to execute a binary file (having execute permission), the new process inherits the real and effective uid number of its parent.

However, when *setuid* permission is set on a binary file, the process being run from this file does not inherit its effective uid number from its parent process, but it is set to the identity number of the owner of the binary file.

Setting the *setuid* permission on a binary file by the owner, (s)he enables everyone to run the program with the power of its owner. Such process being run from the setuid binary, however is able to find out the real identity of the user according to the own real uid number and may choose its behaviour respectively when not being run by the owner.

This concept was raised to enable common users executing some operations that need the effective user id of the system administrator, because they try to access protected system objects. Thanks to setuid permission, common user may for example change its own system authentication password or try to ping some network address.

Final concept used in traditional UNIX access control is the *stickybit*. It was introcuded in order to strengthen the security of files in directories shared among several users. In Linux, the sticky bit has defined semantics only on directories and when set, it protects files within that directory against deleting or renaming by anyone except from the system administrator, the owner of the file or the owner of the directory. Without sticky bit, anyone with write permission for the directory could make these operations.

Access control lists

Access control lists[3] enable defining more fine-grained discretionary access control rules. They are also bound to objects of a file system and semantically define a superset functionality when comparing with the traditional Unix permissions.

Access control list consists of the set of entries. Each entry is a pair of a tag type and a set of permissions (read, write, execute). Respective permissions are granted on the given object to the user or group specified by the tag type.

Three categories of tags (`ACL_USER_OBJ, ACL_GROUP_OBJ and ACL_OTHER`) correspond to traditional permissions. Remaining two tag types (`ACL_USER, ACL_GROUP`) need to be more closely specified with a tag identifier identifying the user or the

group. This way it is possible to explicitly assign permissions to the specific user or group on a particular object regardless of its ownership.

It is also possible to assign the default access control list to a directory specifying the default value of ACL when creating a new object within the directory.

In theory, there could arise a conflict between permissions set in access control list and traditional permissions, but Linux prevents this by simultaneous changes to both permissions when change on one of them is requested.

1.2 Mandatory access control

Mandatory access control[4] is a stricter form of access control by which operating system constrains the ability of subjects to perform operations on objects. The operating system keeps track of various security attributes on both objects and subjects. When an operation is requested, the operation system kernel examines the security attributes of concerned entities and makes a decision according to the set of authorization rules.

These authorization rules are also called the security policy. The policy is managed centrally by an administrator of the security policy. In contrast to discretionary access control, where the policy may be overriden by users alone, allowing them to grant access permission to objects they own, in mandatory access control this is not possible. Policy is therefore system-wide and guaranteed to be enforced for all users.

Mandatory access control use originates in a military and government enviroment, which are the typical multi-level security organization structures. Such enviroment operates with documents (objects) on various sensitivity levels and people (subjects) with different levels of trust are employed there. Therefore strict central access policy had to be adopted to prevent data corruption or escape.

In the next sections, I will mention a brief overview of basic principles of some existing mandatory access control policies implemented for Linux and also the core of Two-Dimensional Labelled Security Model with Partially Trusted Subjects, whose Linux implementation is the main goal of this thesis. The formal definition of information flow policy of this model is described in the next chapter. For more

information regarding this model look at [1].

1.2.1 SELinux

Security-enhanced Linux[5] is the operating system mechanism enabling support for a wide range of mandatory access control policies.

Originally it was developed by the United States National Security Agency, later on released for an open source development and incorporated into the mainline Linux kernel as a patch. SELinux functionality strongly influenced the design of Linux security modules framework for writing user defined access control policies. When it was released, SELinux was rewritten using this architecture.

SELinux may enforce mandatory access control policies with concepts such as multilevel security, role-based access control and domain-type enforcement. The main goal is to grant programs only the minimum privileges they need to do their jobs while restricting other privileges that could be misused intentionally or by program bugs. This is everything done on top of the discretionary access controls.

SELinux assigns information called the *securitycontext* to all subjects and objects. Security context consists of three parts: *user*, *role* and *type*. For subjects we speak of a *domain* in case of the *type*.

In most defined policies used, the *user* and the *role* parts are neglected and access control is based only on *types*. This concept is called the domain-type enforcement. Only operations explicitly stated in the policy between pairs of domain-type are permitted. The classification of operation types is much richer than traditional read-write-execute model. The idea behind is to divide system objects into separated well defined areas called sandboxes by assigning the same *type* to all objects in a sandbox and limit a particular subject to several operations on a particular sandbox. Due to the fact that some complex processes such as init influence wide parts of the operating system, the domain-type enforcement policies would have to define too large sandboxes and that would lead to losing their sense. Therefore transitions of subjects between various domains is allowed in the stated transition rules.

1.2.2 Two-Dimensional Labelled Security Model with Partially Trusted Subjects

In this section I will describe the principles of this security model realizing mandatory access control with multi-level security. The formal definition of allowed information flow in this model is concerned in the next chapter. More information about this model can be found at [1]. The rest of this thesis contains implementation details of this model as a security module for Linux kernel, some additional concepts added in order to keep things working while maintaining safety and the deployment guide and tips for a correct configuration of the policy.

The model recognizes similarly as all previously defined security models in Linux two kinds of entities:

- *objects* - passive entities, carriers of information, for example files, sockets,pipes, shared memory segments,...
- *subjects* - active entities, processing information, these are processes and threads

The model is focused on securing two security attributes:

- $confidentiality$ - level of protection against unauthorized reading
- *intgrity* - level of protection against unauthorized modification

Defined access policy is based on the two models designed for securing two attributes mentioned above.

Bell-Lapadula model classifies existing objects into multiple confidentiality levels according to the information they hold. Each normal nonprivileged subject in the modeled enviroment has a confidentiality level associated which limits operations it can execute. The general rule in Bell-Lapadula model is no read-up and no write-down. This means that subject may not read information on a higher security level and may not write to a less secure object, which would lead to information declassification.

Biba model was designed to overcome disadvantages of Bell-Lapadula model concerning integrity. Similarly it classifies information into integrity levels according to its thrustworthiness. We may naturally consider information which is stricly protected against modification to be thrustworthy. A general rule for the integrity protection is no write-up and no read-down. This means that an unprivileged subject may not modify information above its security level and is forbidden to read information classified with a lower level in order to secure the subject against being spoiled by untrusthworthy data.

Apart from just focusing on security attributes viewpoint,the Two-Dimensional Labelled Security Model also watches the user identity of interacting entities. It secures highly confidential data of a user against reading by another user's subject. Similarly the data with strict integrity protection may not be modified by any subject of another user.

Bell-Lapadula and Biba model are suitable when we are demanding a high level of security and enforcing multi-level security policy, but in general operating system such as Linux there are too many scenarios requiring information flow policy exception from one of the models.

The traditional solution of handling exceptions from policy is deploying *trusted subjects* allowed to violate the policy. This solution would however require too many trusted subjects. In order to keep the system in a secure state, we must be sure, that each trusted subject tries to execute only the claimed functionality (which needs exception in the policy) and also that the program does not contain implementation errors allowing it to misbehave, which could have fatal security consequences with respect to the buggy subject's privileges.

Making such strong assumptions about all trusted subjects needed in the operating system enviroment is practically unreal, as a result lowering global security. This problem was overcome in the described model by thinking the third type of subjects out - *partially trusted subjects*.

Before adopting partially trusted subjects, it is necessary to enhance the security context of objects in the model by another security attribute - object *label*. The label may be thinked of as an identifier of the domain to which the object with its

data belongs. Partially trusted subject is restricted to operate exceptions out of the information flow policy just on objects with labels, that are configurated for a given subject. The configuration of labels comprises of specifying a set of approved labels for each combination of the security attribute (confidentiality, integrity) and the operation (read,write) concerned. Violating the policy by partially trusted subjects within these given input/output labelled objects may be further resticted by defining minimum/maximum bounds of the object's security level (confidentiality, integrity) needed to allow violating the policy.

To sum up, Two-Dimensional Labelled Security Model distinguishes three types of running subjects considering the level of trust[1]:

- *untrusted subjects* - not trusted to enforce information flow policy
- *trusted subjects* - trusted to enforce information flow policy with intended exceptions
- *partially trusted subjects*
  - trusted not to transfer information from a dened set of objects at a higher condentiality level to a dened set of objects at a lower condentiality level in a way other than the intended one
  - trusted not to transfer information from a dened set of objects at a lower integrity level to a dened set of objects at a higher integrity level in a way other than the intended one
  - not trusted to transfer information between any other objects

Chapter 2

Information Flow policy

This chapter introduces The Formal definition of the information flow policy of Two-Dimensional Labelled Security Model with Partially Trusted Subjects security model.[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.

[1] This chapter is a part of the [1]

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

$$\begin{aligned} \mathbf{read}(S,O) \Leftrightarrow & [CR_S \geq C_O \vee (CRL_S \geq C_O \wedge L_O \in CRLS_S) \\ & \vee (C_{appr} \geq C_O \wedge \mathbf{UserApprovedRead}(S,O))] \end{aligned} \tag{2.1a}$$

$$\wedge [IR_S \leq I_O \vee (IRL_S \leq I_O \wedge L_O \in IRLS_S)] \tag{2.1b}$$

$$\wedge [U_S = U_O \vee C_O \leq C_{shareable}] \tag{2.1c}$$

$$\wedge [U_S = U_O \vee U_O \in IRUS_S \vee IR_S \leq I_{shareable}] \tag{2.1d}$$

where **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 **write**(S, O) is true, where

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

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

$$\begin{aligned}\textbf{create}(S,P) \Leftrightarrow &\textbf{read}(S,P) && \text{(2.3a)}\\ \wedge &\textbf{write}(S,P) && \text{(2.3b)}\end{aligned}$$

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

$$C_O := \begin{cases} CWL_S & \text{if } L_P \in CWLS_S \\ CN_S & \text{otherwise} \end{cases} \quad \text{(2.3c)}$$

$$I_O := \begin{cases} IWL_S & \text{if } L_P \in IWLS_S \\ IN_S & \text{otherwise} \end{cases} \quad \text{(2.3d)}$$

$$L_O := LN_S \quad \text{(2.3e)}$$

$$U_O := U_S \quad \text{(2.3f)}$$

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

$$\begin{aligned}\textbf{delete}(S,O,P) \Leftrightarrow &\textbf{read}(S,P) && \text{(2.4a)}\\ \wedge &\textbf{write}(S,P) && \text{(2.4b)}\\ \wedge &\textbf{write}(S,O) && \text{(2.4c)}\end{aligned}$$

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:

$$C = \{0, 1, 2\}$$
$$I = \{0, 1, 2\}$$
$$C_{appr} = 1$$
$$C_{shareable} = 1$$
$$I_{shareable} = 1$$

with the meaning of the confidentiality levels:

$$0 - \text{public},$$
$$1 - \text{C-normal},$$
$$2 - \text{C-sensitive},$$

and the meaning of the integrity levels:

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

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 $\mathbf{reclassify}(S, O, c, i)$ is true, where

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

- A subject D (the debugger) may use the debugging interface to debug a subject S if $\mathbf{debug}(D, S)$ is true, where

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

- A subject S may send a signal to a subject R if $\mathbf{maysignal}(S, R)$ is true, where

$$\mathbf{maysignal}(S, R) \Leftrightarrow CW_S \leq CR_R \tag{2.9a}$$
$$\wedge IW_S \geq IW_R \tag{2.9b}$$
$$\wedge U_S = U_R \tag{2.9c}$$

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

$$\mathbf{setCN}(S, c) \Leftrightarrow c \geq CW_S \tag{2.10}$$

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

$$\mathbf{setIN}(S, i) \Leftrightarrow i \leq IW_S \tag{2.11}$$

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

$$\mathbf{setCR}(S, c) \Leftrightarrow c \leq CR_S \tag{2.12}$$

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

$$\mathbf{setCW}(S, c) \Leftrightarrow c \geq CW_S \tag{2.13}$$

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

$$\mathbf{setIR}(S, i) \Leftrightarrow i \geq IR_S \tag{2.14}$$

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

$$\mathbf{setIW}(S, i) \Leftrightarrow i \leq IW_S \tag{2.15}$$

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

$$\mathbf{setCRL}(S, c) \Leftrightarrow c \leq CR_S \tag{2.16}$$

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

$$\mathbf{setCWL}(S, c) \Leftrightarrow c \geq CW_S \tag{2.17}$$

9. A subject S may change IRL_S to i if $\mathbf{setIRL}(S, i)$ is true, where

$$\mathbf{setIRL}(S, i) \Leftrightarrow i \geq IR_S \tag{2.18}$$

10. A subject S may change IWL_S to i if $\mathbf{setIWL}(S, i)$ is true, where

$$\mathbf{setIWL}(S, i) \Leftrightarrow i \leq IW_S \tag{2.19}$$

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

Chapter 3

Linux security modules

Tha aim of this chapter is describing a security framework enabling implementation of various user defined security models of mandatory access control. I will introduce key design elements of the framework, give an overview of controlled kernel structures and think about feasibility of Two-dimensional labelled security model with partially trusted subjects implementation for Linux using Linux security modules architecture.[6][7]

3.1 The history

The need for a general purpose security framework in Linux kernel raised from several reasons:

- kernel community realized that conventional discretionary access control in Linux is not secure enough for various scenarios
- several independent security projects were running, to use any of them, a kernel patch had to be installed, patches interfered with various parts of existing kernel code, this approach could easily bring incostistencies during new kernel releases, patches had to be applied for each new kernel installed, due to these circumstances, a massive deployment of any of new security mechanisms was not practicable

- the leaders of several projects emphasized the need of a flexible access control architecture support in the mainstream Linux kernel, since Linus Torvalds[1] did not want to favor any particular projects prior others, he suggested building a general access control framework with the infrastucture enabling kernel modules to implement security functions

Dynamic loadable kernel modules existed prior to these activities, mainly for loading device drivers and filesystems. A problem with their usage for security implementation was that tke kernel did not provide them infrastructure to mediate access to kernel objects. Therefore implementing security by kernel modules falled down to interposition system calls requested by user space processes to the kernel. This approach is however not sufficient for required security functions.

The constraints put on Linux security modules framework design by Torvalds were following:

- truly generic, using a different security model is just a matter of loading other kernel security module
- simplicity and low performance overhead
- ability to implement existing POSIX.1e capabilities as an optional module

Further debates were held about the scope of the security framework. Some projects were interested in security auditing and system virtualization apart from access control. The flexibility of access control was debated as well. Most of the models were interested just in further restricting access on top of the existing DAC, but some models required also the ability to grant prileges even if DAC logic denied the operation.

3.2 The design of Linux security modules

The core idea of Linux security modules design is to mediate access to internal kernel objects. Provided security module should give answers to question whether

[1] Linux kernel project coordinator

currently running subject S may execute a given operation on a kernel object O.

This is done by placing *hooks* into the kernel code just before the place where original code would permit access to an internal kernel object and would perform some action on that object. The responsibility of a given hook is to call the function provided by the loaded security module that manages this kind of action on a particular kernel object type. The hook should provide all security context parameters to the judging function that could be relevant in deciding according to LSM designers. The module function may either permit access or deny it by returning specific error number.

The requirement of simplicity of LSM[1] design caused that most of the hooks are restrictive. This means that a security module is asked to decide whether to grant or reject access only if the access would be granted by DAC access control. In the opposite case, the access is denied before the security module function is even consulted. This is caused by too many access denying shotcuts called early in the kernel code when some error conditions are detected by traditional access control semantics. To be able to consult all these situations with security module, too many authoritative hooks (their decision is final) would have to be inserted into the code, that would also rapidly increase the number of hook functions implemented by the security module realizing particular policy.

However to enable implementing POSIX capabilities policy as a security module, its logic requires possibility to grant access despite DAC[2] deny to some coarse grain extend. This is realized by a permissive hooks calling the *capable*() function realized by the loaded module. The module is asked to judge by this function whether currently running process has a capability to override some discretionary controls. Therefore these permissive hooks are usually coupled within code with some basic discretionary access checks and enable module to override them.

[1]Linux Security Modules
[2]Discretionary access control

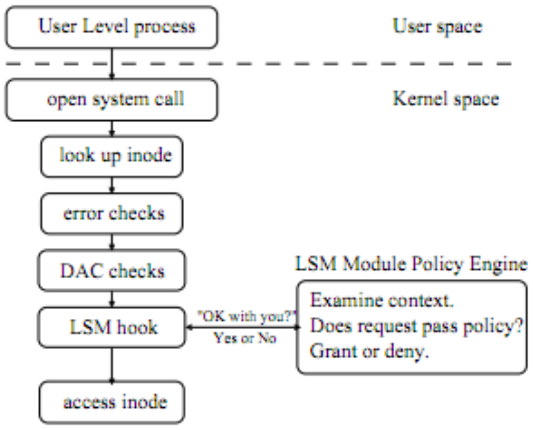

Figure 1: LSM Hook Architecture

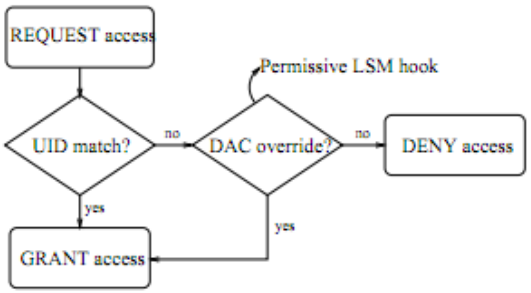

Figure 2: Permissive LSM hook. This hook allows the security policy to override a DAC restriction.

A very important concept required by many policies and therefore included in the Linux security modules design is the ability to bind security context to protected kernel objects. LSM provides this by attaching opaque security fields to their respective structures. These are realized by pointers of *void* type. LSM lets all work done with the content of these fields to administration of the loaded security module and provides usually allocate, deallocate and update hooks for all protected kernel objects, which are suitable place to initialize, free and change the security context of objects by module according to the realized policy. Also locking access to these field against concurrent access is in hands of the loaded module.

Stacking of several security modules is very limited by the LSM architecture. LSM enables stacking of several modules, where the first one act as a primary module a it may enable other module to register itself and provide its implementation of hook functions. However, the decision to call functions of the secondary modules and evaluate their results is fully left to the primary module. This approach requires stacking modules to have knowledge of each other. Automatic stacking of modules is not possible. The major reason is that LSM provides just a single pointer to security fields, so that stacking modules would simultaneously rewrite it. There would also have to be the control module on top of the stacking modules calling each registered module's functions and combining their result and assigning place in kernel objects for modules on their request.

However, stacking with the built-in POSIX.capabilities module works automatically, because capabilities have the space in kernel structures, they work with, assigned separately from the opaque security field. If a hook function is called while a user defined security module is loaded, then LSM checks whether this module implements this hook function. In case it does not, then the capabilities hook function is called. In the opposite case, the security module's function is called and it may or may not ask the capabilities module for its access control decision.

When a security module wants to enforce the policy it implements in the system, it has to registrate itself to the kernel. It is done by calling the *register_security* function passing a *security_ops* structure to the LSM framework. The security_ops structure identifies a name of the policy, that has to be unique and should define individual functions realizing the logic of implemented policy bounded to the respective hook functions. This means assigning the pointers to functions to the formal names of hooks. Signatures of implemented functions has to coincide with hook functions.

After booting the system, the LSM framework is initialized with the dummy hook functions realizing the traditional super-user DAC semantics. When a security module tries to regiter itself to LSM, the dummy functions are replaced in the global security operations table with the passed security functions realized by the module . Hook functions not defined by module stay original.

3.2.1 Protected kernel objects and operations

I will briefly mention here the overview of kernel objects that may be secured by the LSM architecture and important mediated operations on them to catch on the power of Linux security modules framework.

Mediated operations according to their goal may be divided into the two main categories. Some operations execute access control granting or denying the specific access to kernel structures, they realize the security policy logic.

On the other hand, the purpose of some operations is to manage internal security context of kernel structures, that means initializating, deallocating, changing structures.

Finally there are hybrid hook functions, which either make requested changes to the security context and return success or deny access.

I will list important mediated kernel structures and controlled operations and tell which known entities from the user space of the operating system are associated with them.

inode

Inode is a basic data structure holding information about objects of the file system. This include files, directories, symbolic links and special objects such as sockets, pipes and devices. It includes metadata about these objects such as access mode in DAC, owner, size, parent file system and identifies uniquely the object within its filesystem by the inode number. It does not keep data, name or path of the object. What is important from security context, LSM added a security field *i_security* to the inode kernel structure.

The main acces control hook for the inode structure is *inode_permission*. The key hook for determining inode's security label is *d_instantiate* called when filesystem object associated with the inode is being inserted to the root file system.

task_struct

The task_struct represents a schedulable task within kernel, e.g. a process or a thread. It holds informations about a task such as process id number, priority, parent task, children tasks etc. From the security context the most important is a pointer to the *cred* structure holding security credentials of the task, such as real uid, effective uid, process capabilities and *security* opaque field for assigning specific policy security context.

LSM enables the control of sending a signal to the task, tracing it, working with the *security* opaque field, transferring credentials of a task to another task and many other operations.

linux_binprm

The linux_binprm kernel structure represents a program being executed during the *execve* operation. A security context of the executed program is kept within the same *cred* structure as for tasks.

There are various hooks related to the linux_binprm called at different stages of program execution. These hooks may forbid this activity or may initialize the new security credentials of the executed program (based on the credentials of the currently running executing process for example).

super_block

This kernel structure represents a file system. It provides the *s_security* opaque field for storing arbitrary security context defined by the loaded security module.

The most important super_block hook mediates mounting, unmounting a file system, reading its statistics or passing mount options during the mount process.

file

File structure corresponds to the opened file. It allows a security module to bind security information separately from the respective inode security field. While opening a file, its inode permission hooks are consulted. The file strucuture specific hooks are usually used for revalidating the read/write permission at each executed operation on an opened file, mediate file locking or various operations to an opened file through the system call fcntl().

kern_ipc_perm

This structure represents the security credentials of various inter-process communication objects. The same structure is bounded to all of the objects, these are semaphores, shared memory segments or message queues.

sk_buff

This kernel structure corresponds to the packet. LSM has also added the security field there to be able to preserve the security information across all network layers the packet flows through.

Applicaton networking layer is mediated by socket related hooks. Socket kernel structure does not have separate security field added by LSM, the security context may be stored in respective inode's i_security field. Socket security hooks may differ their behaviour based on recognized communication protocol from the socket structure.

3.3 Feasibility of implementing Two-Dimensional Labelled Security Model with Partially Trusted Subjects by LSM architecture

In this section, I will try to outline that it is possible to implement this security model in Linux with the use of the Linux security modules architecture.

The first precondition is that Two-Dimensional Labelled Security Model does not need to override discretionary access control, what meets the design of LSM.

First I will analyze which kernel structures correspond to abstract entities in the information flow policy of the model.

Subjects comport purely with the in operating system which are matched with the task_struct inside the kernel.

Objects in the security model may correspond to files (directories), pipes, sockets, filesystems, packets, semaphores, message queues, shared memory segments and individual messages. These abstract objects are modeled by respective inode, super_block, sk_buff, kern_ipc_perm and msg_msg structures within the kernel.

LSM added security fields to all these structures, so it is possible to bind security attributes defined in the policy to all abstract objects and subjects. Linux security modules does not limit in any way the size or the form of bounded attributes to the structures.

Now I will try to show that operations on abstract entities defined in the informatin flow policy of model may be mediated using the Linux security modules architecture. For each operation controlled in the model, I will list respective hook functions mediating the operation. Some hooks listed may cover the functionality in a larger scope than respective abstract operation (for example based on the input parameters). I will not mention the details of hooks here, the important ones will be described closer in the implementation chapter.

For operations such as read and write, having very general semantics, I will list only the basic hooks. In fact these operations are covered in a vast number of hooks, since every hook mediating access to internal structure's metadata may represent a hidden communication channel. I will only list hooks controlling access to the actual data.

- *read* - inode_permission, ipc_permission, msg_queue_msgrcv, shm_associate, sem_associate
- *write* - inode_permission, ipc_permission, msg_queue_msgsnd, shm_associate, sem_associate
- *create* - inode_create, inode_mknod
- *delete* - inode_unlink
- *debug* - ptrace_access_check
- *signal* - task_kill
- *reclassify* - inode_setxattr
- *change subject's attributes* - setprocattr

Another important thing is storing the security attributes of permanent objects. Since modern filesystems used in Linux have extended attributes support and LSM mediates also managing those attributes, this condition is fulfilled.

The aim of this section was not the formal proof of feasibility of the Two-Dimensional Labelled Security Model implementation using LSM in Linux. It is

not even doable because of several reasons. It is difficult to accurately specify the scope of information protection required when deploying the formal security model onto the living operating system of such complexity. Even if this was determinable, there would have to be the formal proof of mediating every watched operation by Linux security modules architecture.

The real goal of this section was rather showing, that there are not major obstacles in the logic the of security policy that would be uncatchable by LSM design.

Another feasibility argument is the implementation of the security model using SELinux.[8] Since SELinux is currently build on the LSM architecture, it should be feasible to make a standalone security module realizing the security policy.

3.4 Current state of Linux security modules

There has been changes in Linux security modules API in 2007, influencing rapidly developing and a deployment process of security modules, although not changing the mediating power.[9] The patch coverted LSM into the static interface. This change disabled the possibility to realize a security policy as a dynamic loadable kernel module, what aims a little bit against the original idea of LSM. There were several arguments reasoning this change. First,the LSM loadable interface was expected as an ideal entry point for inserting malicious rootkit code into the kernel. This could have been done especially with the closed third party modules, which pretended to realize the security policy. The second argument was the nonexistance of real security project necessary requiring loadable and unloadable interface.

The final solution does not export the LSM interface for modules use and incorporated several most popular security modules into the mainline kernel.

This change has brought some complications for my implementation. The security policy has to be implemented as a kernel patch, what means unevitably recompiling the kernel after each change in the code, creating and setting boot image and rebooting the system to test new implementation.

Chapter 4

Implementation

This chapter will continously describe the steps of implementing the Two-Dimensional Labelled Security Model with Partially Trusted Subjects security model in Linux using LSM. There have raised several points, where implementing the policy logic is not straight forward and requires more attention. I will always mention several possible solutions at this points and clarify suitability of my decision. Finally I will closely describe the new concepts added to the policy at this points, their semantics and rules for their maintaining.

4.1 First tries - Implementing dummy policy

First step I tried to do was test out some sample security policy implemented by LSM. First attempt was with the root_plug policy used to demonstrate the power of LSM during its release, which prevented running tasks with the effective id of a system administrator unless the specific usb drive is plugged in. However I did not manage to even compile this module. At first I thought it is due to the changes of LSM hooks or usb device interface since then. However I got the same results when writing my own dummy security module that prohibited reading of a file with some hardcoded specific name.
After that I studied more closely the LSM kernel code[10] and I realized that it is not exporting any symbols to the userspace for use in kernel modules. Finally I found

out from kernel developers' mail logs the change in LSM API made static. Since the web page of original LSM development group no longer works and all materials relating LSM are dated to its first deployment it took me a while to find this out.

At this stage I implemented the dummy policy as a kernel patch. To test it out, it was necessary to recompile the kernel with the patch added, install the new kernel image to the boot menu with security boot option set to the name of the dummy security module and reboot. For testing I tried various operations in the system and compared their results with the expected outputs. Another approach to finding out closer what is done within the kernel is to place control print outputs at the prominent places and watch kernel logs during the tests.

Finally, I realized experiments with the kernel compilation after changes made to the security module to speed up the development and test cycles. The result is approximately 5 minutes of time needed to recompile the kernel with the changes made only to the security module files.

4.2 Basic structures and operations

This section describes the implementation of structures and operations on the abstraction level of the information flow policy. These operations either perform access controls or manipulate with respective structures.

The implemented policy is recognized by the *pts* name within the kernel, related to the partially trusted subjects concept in the model. Entities mentioned in this chapter can be found at the header file *pts.h* included in other files implementing the security model. To improve the development and testing of this part, I implemented and tested it as a kernel module first, since it does not yet use the LSM achitecture. Implemented parts were then moved to the kernel patch for the security policy.

Following two structures represent the security context of objects and subjects:

```
struct subject_pts {
        int cr_s;
        int cw_s;
        int crl_s;
        int cwl_s;
        struct list_head crls_s;
        struct list_head cwls_s;

        int ir_s;
        int iw_s;
        int irl_s;
        int iwl_s;
        struct list_head irls_s;
        struct list_head iwls_s;

        int cn_s;
        int in_s;
        char ln_s[PTS_LABELLEN];
        uid_t u_s;
        struct list_head irus_s;
        struct list_head cwus_s;

        struct mutex lock;
};

struct object_pts {
        int c_o;
        int i_o;
        char l_o[PTS_LABELLEN];
        uid_t u_o;

        struct mutex lock;
};
```

I modelled confidentiality and integrity levels as an integer type, admitting a possible use of negative values, what was later used for the model expansion. Labels are character arrays with a static length defined as a value of the macro object $PTS\_LABELLEN$. Using the static array is easy to manage and with a reasonable maximum length set is not limiting. Identifier of a user is of a type uid_t which is used for numbering of users in the rest of kernel as well.

The label and user id sets are implemented as linked lists. Assuming small cardinalities of these sets during the real usage, the linear time of searching does not matter. The elements of these linked lists are of a type $label\_item$ or $user\_id\_item$, which are the wrapper structures for label and user id number. These structures have to contain member of a type list_head enabling navigation within linked list. Kernel provides a way to dereference the particular list_head member and get its

wrapping structure, i.e. object of label_item or user_id_item type.

4.2.1 Policy parameters

All initial paramaters of the policy are defined as macro objects with their respective values set prior to patching the kernel. There is a variable corresponding to each parameter, whose value is initialized with a value of the respective macro object directive.
Example of default confidentiality level of object set to 1:

```
#define DEF_C_O 1
int def_c_o = DEF_C_O;
```

My implementation sat out with such a static solution for defining parameters permanently before the deployment. This solution should be sufficient in most cases, where the semantics of security levels is clarified and fixed prior the security module deployment in the system. Redefining parameters in a situation, where the attributes of permanent objects are set according to the former manners could bring an unwanted and hardly catchable behaviour in the new settings.

Despite this, the dynamic setting of the policy could be easily realized by passing them as boot options of a currently loaded kernel and rewriting particular variables holding the parameters.

The described parameters include permitted ranges for confidentiality and integrity levels, default values for invidual members in *object_pts* and *subject_pts* structures and default values for the three system constants defined in the information flow policy.

I used the following settings:

- C and I ranges from 0 to 2
- all default values concerning C and I are 1, including the system constants
- default label is set to an empty string
- default label and user id sets in subject_pts are empty

4.2.2 Operations

There are set of methods realizing allocating, deallocating, resetting and cloning the basic structures.
Function signatures for the subject_pts structure:

```
struct subject_pts *new_subject_pts(gfp_t)
void del_subject_pts(struct subject_pts *)
void reset_subject_pts(struct subject_pts *)
struct subject_pts *duplicate_subject_pts(struct subject_pts
*,gfp_t)
```

Allocating creates a new structure with the default values set, the input parameter of a type gfp_t is the required atomicity of memory allocations and is one of the arguments used for allocating memory within the kernel. It had to be added due to the fact, that allocating security context of subjects is required also during some specific states of system(booting for example), when specific memory allocation schemes are used.

Resetting takes an existing structure and changes it to the default state. Duplicate operation takes an existing security structure and make its copy.

Another operation for creating new objects is *new_object_pts_by_subject*, which simulates creation of a new object by the subject with its security context passed as an argument.

Changes of basic structures

Changes of security levels are done by direct assignment of an integer to the changed value. Setting a new label is done with helper function *normalize_label*, which takes care of the maximum length of label allowed.
Various helper operations on sets are defined explicitly. I have created macro function templates for general operations(add, delete, clear, contain) on sets. These macros require the actual type of objects within set to implement the operations create, delete and compare. I have realized these operations for the used types *label_item* and *user_id_item* and created general operations(add, delete, clear,

contain) for sets containing these types by appropriate assigning parameters to the macro function templates.

Text serializing/deserializing operations

Since security attributes should be adjustable by authorized subjects from the user space and they will be passed in the natural text form, it was necessary to define equivalent textual representation of object and subject structures' values. The representation may be of two slightly different types: static showing the current state of security attributes and a dynamic representation requiring changes of security attributes. These have the equal syntax for objects.

Both representations consist of clauses separated by semicolons. Each clause relates to one member of a security structure.
Clauses have the following form: *member_name operator value*
In case of a static representation and non-set clauses of dynamic representations the operator is always = . In textual change requests for sets, also + = and − = operators are allowed. These require adding(removing) value following them to(from) the respective set in the structure. Value in set clause consists of comma separated single values.

Since only letters of the alphabet and digits are allowed as characters of labels, this representation is unambiguous.
Example of a default object textual representation: $c\_o = 1; i\_o = 1; l\_o =;$

User identity attributes within subjects and objects structures denoting the owner are derived attributes, they are already present in the kernel structures and their values is copied to defined security structures from there. Mechanisms for getting and setting their value are already present in the operating system(system calls setuid and chown) and therefore are not included in the textual representation.

To be able to effectively work with the textual representation and make conversions to/from the security structures following functions were defined:

```
int valid_subject_pts_string(char * s)
int string_to_subject_pts(char *s, struct subject_pts *sub, int
```

```
resetHeritable)
char *subject_pts_to_string(struct subject_pts *sub, int *size)
```

The first function validates the passed string against the textual representation format. The second function changes security attributes of the subject structure according to the string *s*. The third argument will be explained later. The last function returns the textual representation of current security attributes of *sub* and sets *size* pointer value to its length.

Analogical functions exist for the object security structure.

Control operations

Final stage of defining this abstract level of security policy are the operations realizing access control. The most significant is

```
int pts_access(struct subject_pts *sub,struct object_pts *obj,int request)
```

deciding whether subject *sub* may execute operation *request* on object. The return value is 0 in case of granting access or $-EPERM$ error code in case of a denial. The coding of the third argument is taken from the permission model used in kernel. Individual bits of this integer means following basic operations:

```
MAY_EXEC 1
MAY_WRITE 2
MAY_READ 4
MAY_APPEND 8
MAY_ACCESS 16
MAY_OPEN 32
MAY_CHDIR 64
```

Since implemented policy recognize only read and write basic operations, I defined my own permission integer bits this way:

```
#define MAY_READ_PTS (MAY_EXEC | MAY_READ | MAY_ACCESS | MAY_OPEN)
#define MAY_WRITE_PTS (MAY_WRITE | MAY_APPEND )
```

There may be passed several operations to the function through request argument, therefore function computes all operations the passed subject may do with the object according to the information flow policy and the result is logically anded with request. When the final result is equal to the original request, it means all requested operations are permitted and the access is granted.

4.3 Implementing basic LSM hooks

Now it is time to implement the first trial version of policy, which uses defined security structures and access control operations. The first version works just with the default values of attributes in the security structures.

The init process is initialized with the default subject_pts security context and because it is an ancestor of every running process and changes of a subject security context are not yet implemented in this version, every process runs with the default subject attributes.

Analogically, all objects are either created dynamically by subjects with the default attributes or belong to permanent objects, getting their security attributes from stored extended attributes which use in the policy will not be supported yet in the initial version.

Every requested operation enabled by DAC sent to the initial policy hooks therefore should be allowed.

In following sections I will focus especially on tasks and filesystem objects, whose security is maintained by inode hooks. Following initialization inode hooks are implemented:

```
int inode_alloc_security(struct inode *inode)
void inode_free_security(struct inode *inode)
int inode_init_security(struct inode *inode, struct inode
*dir,char **name, void **value, size_t *len)
```

The first hook binds the object_pts structure derived from the security context of the currently running subject to the inode i_security field during the inode initialization. The second hook frees memory occupied by the security context of the

currently destroyed inode. The third hook enables setting of extended attributes concerning security that the currently loaded policy wants to assign to the initialized inode. In our case it will be single security attribute, I called it *PTS*. The hook function assign this to the name argument and fills value with the textual representation of inode's security attributes, len argument holds its size.

Now it is time to implement initialization hooks for tasks. These are:

```
int cred_alloc_blank(struct cred *cred, gfp_t gfp)
void cred_free(struct cred *cred)
int cred_prepare(struct cred *new, const struct cred *old, gfp_t
gfp)
void cred_transfer(struct cred *new, const struct cred *old)
```

The first hook should allocate sufficient memory for the security context assigned to the initialized task. In our case it binds the subject_pts structure with default attributes to the passed credentials' security field. The second hook frees memory holding the security context of the terminating process. Remaining two hooks realize transition of security credentials, they create the copy of subject_pts structure in old credentials and assign it to the new ones.

Finally following access control hooks are implemented:

```
int inode_permission(struct inode *inode, int mask)
int inode_create(struct inode *dir, struct dentry *dentry, int
mode)
int inode_mknod(struct inode *dir, struct dentry *dentry, int
mode, dev_t dev)
int inode_unlink(struct inode *dir, struct dentry *dentry)
int my_inode_rename(struct inode *old_inode, struct dentry
*old_dentry, struct inode *new_inode, struct dentry *new_dentry)
```

In all these hook implementations, the security contexts of participating object(s) and subject are retrieved(in the time of calling these hooks, they are already set) and they are passed to the pts_access function with the correct request according to the information flow policy. In case of the inode structure, the security context

is held in i_security field, in case of dentry it may be navigated to the corresponding inode and retrieved from there. Subject security context is gained by calling current_security().

The first hook is general access control, it passes directly the mask request to the pts_access with the exception of MAY_EXEC request on a directory, which is automatically granted in our security model(when accessing a filesystem object in DAC execute permission is controlled through the full path to the object, this is kept in LSM, however our model does not explicitly recognizes MAY_EXEC).
The second hook and third hooks correspond to creating inode at the passed dentry, with the dir inode of its parent directory. This is judged according to the create operation in the policy. The difference between hooks is that the second hook is called when creating a regular file, the third is called for special objects as pipes and sockets.
The fourth hook is used for controlling the delete operation.
The final basic control hook is used for control when renaming dentry. What is unusual is that if the new renamed dentry already exists, this object is deleted without calling inode_unlink or inode_permission hook. I simulated rename operation as requiring both read and write access to the inode corresponding to the old dentry and in case the new dentry existed before also requiring write access to its inode.

Kernel hooks implemented in this section build the policy infrastructure with static assigning of security contexts just during the initialization of objects. Changes of live objects cannot be done so far. They will be discussed in the next section.

To finish this static infrastructure I will show how object security attributes may be specified non-default during their initialization using following hook:

```
void d_instantiate(struct dentry *opt_dentry, struct inode
*inode)
```

This hook is called when an inode is being bounded within the filesystem tree to the passed dentry. If the inode belongs to permanent objects, the given opt_dentry is checked whether it has extended attribute used by the security policy set(PTS in our case). If yes, the validity of its value is tested(the textual representation of object_pts) and security context of the inode is set respectively.

In case of objects created dynamically after every system start, this hook provides also suitable place for unified initialization security labeling of objects based on their parent filesystem type(for example if we want to assign nondefault attributes to objects of a device filesystem).

4.4 Hooks serving security attributes changes requests

The aim of this section is to solve managing changes of security contexts of existing entities in the running system. This means implementing hooks called on change requests, describe the way of communicating required changes from the user space to the kernel, on the other side transfering current security state of attributes from the kernel to the user space for reading.

Finally it is crucial to solve the problem, that only specially authorized trusted subjects under the direct control of a user or the security administrator should be permitted to make those changes.

4.4.1 Transfering objects' security attributes

Security attributes of objects may be mediated from the user space with use of extended attributes supported by modern file systems. Extended attributes are pairs name:value attached to files. They are used to hold metadata related to a file. In Linux, the name part consists of two parts, the namespace and actual name(null terminated string) joined with comma. Currently four namespaces are recognized: user, trusted, system(storing ACLs) and security. For implementing our policy the security namespace is important.

Used extended attribute's name is defined by the following macros, where $XATTR\_SECURITY\_PREFIX$ is already defined by the kernel as *security.* and stands for the security namespace:

```
#define XATTR_PTS_SUFFIX "PTS"
#define XATTR_NAME_PTS XATTR_SECURITY_PREFIX XATTR_PTS_SUFFIX
```

Working with extended attributes is supported by the *libattr* dynamic library providing needed functions for user programs. On top of that, there are existing command line programs such as *attr*.

In case of permanent objects, their extended attributes are stored persistently on a hard drive, and in implemented policy during a call of d_instantiate hook, the PTS extended attribute value(if set for the object) is used to determine the security context of the object. For dynamically created objects, their security attributes may be set after their creation within inode_init_security hook.

Following LSM hooks are used for working with extended attributes in security namespace(not controlling access to them yet):

```
int inode_getsecurity(const struct inode *inode, const char
*name, void **buffer, bool alloc)
int inode_setsecurity(struct inode *inode, const char *name,
const void *value, size_t size, int flags)
```

The first hook is called when there is a request for reading extended attribute *name* within security namespace of *inode*. By this hook, loaded security policy may interpret the value of xattribute[1] read and pass the result by $buffer$. In case the policy does not want to interpret the value of the xattr name, it returns $-EOPNOTSUPP$. In this case the actual stored value of this extended attribute is returned(if set) or an 'attribute not set' message.

Example of this usage in our policy:
Let be a value of the PTS xattr for a file f permanently stored as $i\_o = 2$; meaning its integrity level is raised and other values are default. If our policy is loaded and succesful read request for PTS xattr of file f is done, it returns the full textual representation of the security context $c\_o = 1; i\_o = 2; l\_o =$; , but in case no policy is used or policy not interpreting the PTS xattr, then $i\_o = 2$; would be returned.

The second hook is called when a security xatrr *name* is being set to *value*. It provides place to update the security context of an *inode*. In our case it updates the object_pts structure bounded to the *inode* according to the *value* if the *name*

[1] extended attribute

is equal to PTS. It is important to update a permanent xattr value of PTS with the full textual representation of the current object state after accepting the set value. Let's assume for example that a permanent PTS xattr value is stored as $c\_o = 2;$ for an object O and afterwards its owner makes a succesful set query with the value $i\_o = 2;$. We expect that by not mentioning other attributes in the query, the user wants them to remain unchanged. After accepting the query the object's full security context is $c\_o = 2; i\_o = 2; l\_o =;$ If we did not update the stored value of permanent xattribute PTS , after the system reboot the PTS xattr value found for the object O would be $i\_o = 2;$ and therefore we would downgrade its confidentialty to the default value 1, which is an undesired behaviour.

Finally there are access control hooks allowing/forbidding read/write/remove operations on extended attributes. They are called for all xattributes, not just for the security namespace.

```
int inode_getxattr(struct dentry *dentry, const char *name)
int inode_setxattr(struct dentry *dentry, const char *name,
const void *value, size_t size, int flags)
int inode_removexattr(struct dentry *dentry, const char *name)
```

Realization of these is not clear so far and will be discussed later.

4.4.2 Transfering subjects' security attributes

Here I will explain, how it is possible to pass the current security context of a running process from the kernel to the userspace and how to ask for its change from the userspace.

Linux uses the proc filesystem, in default settings mounted at /proc during the boot time. It is used to present various information about running processes and other system information from the kernel to the userspace in a hierarchical structure. For each process there is a directory named after the PID process number holding information about particular process. Process may access own directory structure by using the symbolic link /proc/self without knowing own PID number. It contains

the *attr* directory, with several files which may be read or written to and these events may be maintained by a security policy using following hooks:

```
int getprocattr(struct task_struct *p, char *name, char **value)
int setprocattr(struct task_struct *p, char *name, void *value,
size_t size)
```

These hooks fulfill both access control role and may realize other functionality.

The first hook is called when a file *name* within the attr directory of a process p is read in the userspace. The hook may deny access by returning a negative number with the respective error or set the content of file *name* for reading process by filling the *value* with the content and returning the size of content. In the PTS policy implementation only the file *current* is interpreted, its content is filled with the textual representation of the security context of the subject p at given moment.

The second hook is called when there is a write attempt of the *value* to the file *name* in the attr directory of the process p. Policy may as well deny write or try to make internal changes based on the written value and return the size of an accepted input from the value. In our case, let's assume the *name* is equal to current and write access is permitted. Then the passed value is validated against the subject security context textual representation format and if the result is positive, subjects_pts structure of the task's p credentials is changed respectively.

The decision when to allow changing of the security attributes of a subject is consulted in the next section.

4.4.3 Security context changes policy

I have shown how it is possible to change the security attributes of protected kernel structures from the user space on demand. The important step is how to judge whether the subject requiring change should be authorized to make those changes. The information policy defines which changes should be approved automatically and what changes can be made only by specially authorized trusted subjects under the direct control of the user(I will assume that each user is fully responsible for the objects he owns and all processes running on his behalf and may change their

security attributes with the exception of their owner - but changing the ownership is not done via mechanisms described in last two subsections). In my implementation of the security model, I will tighten up these rules and enable any changes only when done by specially authorized trusted subjects(all original applications running are not aware of my security module, therefore they would not require such changes). I will assume that all support utilities I will implement are well designed and will act as SAT [1] subjects not violating the policy.

Now the big question is how to differ whether the subject asking changes is a SAT subject. Several approaches have come in my mind:

- allowing only one running SAT subject of a certain type(e.g. the type of a SAT subject may be the process manager enabling a user to run a task with arbitrary security attributes), this SAT subject would run as a service being executed early during the boot sequence(set into init scripts), it would register itself as a SAT subject to the security policy(the policy would afterwards denied registering another SAT subject of this type), the subject_pts structure would have to be extended by a flag per every type of the SAT subject, the security policy would set the corresponding flag in the subjec_pts of a registered task, only subject with the set flag could violate the policy according to its privileges as a SAT subject, problems how to communicate with this service(dbus or other form of interprocess comunication would have to be used), what if the service would terminate unexpectedly before system halt?, attention to preventing other applications to register itself as a SAT subject prior to the intended one

- add another extended attribute to the security namespace for marking binary files from which SAT subjects are launched, marking those binaries would be done before the security module deployment, the policy itself would deny attempts to set this extended attributes for other binaries or removing this attribute from the marked binaries, during a binary launching, it would be checked whether this xattr is set, if yes, the added flag in the subject_pts

[1] specially authorized trusted

structure of initialized process marking SAT subjects would be set, during the access control of operations realized only by SAT subjects this flag would be checked for the currently running process

- the third idea is similar to the second, but instead of adding another extended attribute it takes advantage of the POSIX capabilities used in Linux, the capabilities behaviour and its usage for our problem is outlined in the next section

4.5 Capabilities

Traditional Unix discretionary access control distinguished between the privileged processes (effective uid of the root) which avoided all permission checks within the kernel and unprivileged processes which were the subject of permission checks according to their security credentials. This all or nothing privileges model was later divided into smaller distinctive units(capabilities[11]), each representing a single privilege(enabled or disabled separately from others). Capabilities are per process security atributes.

Currently Linux recognizes 35 different capabilities. From the view of the implemented security model especially 4 of them are important:

- CAP_MAC_OVERRIDE - reserved for the implemented MAC [1] policy, privilege to override the policy rules
- CAP_MAC_ADMIN - privilege to modify enforced MAC policy, used by the MAC security administrator
- CAP_CHOWN - privilege to override restrictions of changing the file owner or group, changing the owner of objects is critical operation in our model
- CAP_SETUID - privilege to change one of user ids of a running process, changing the subject's owner should be allowed in our model only to the login managers

[1] mandatory access control

It seems from the suggested use of implemented capabilities, that granting the CAP_MAC_OVERRIDE capability to(and only to) SAT subjects under the direct control of a user would be suitable to implement the managment of security attributes changes by the owner. To see whether it is feasible and how I will closely describe working capability mechanisms and the way to slightly modify them to implement our goal. Each thread has three capability sets, each containing a subset of assigned capabilities. These are:

- permitted - a limiting superset for the capabilities that may be added to the effective set
- inheritable - capabilities, that may be preserved through the execve call running a new program
- effective - these capabilties are used within the kernel for permission checks on privileged operations

When a new program is forked, it gets the copy of capability sets of its parent. To be able to grant capabilities to certain programs, the file capabilities were introduced recently. They are permanently stored as an extended attribute of the binary file within the security namespace.

File capabilities also contain three capability sets:

- permitted - these are automatically moved to the permitted set of a thread being run from this file
- inheritable - they are anded with the inheritable set of an old task calling execve on this file, the result is also moved to the new thread's permitted set
- effective - this is not capability set, but a single bit, if set, the new thread's effective set is equal to the permitted set, otherwise new effective set is empty

Now I will show precisely, how the new thread's capability sets are computed during execve:

```
P'(permitted) = (P(inheritable) & F(inheritable)) |
                (F(permitted) & cap_bset)
P'(effective) = F(effective) ? P'(permitted) : 0
P'(inheritable) = P(inheritable)
```

The capability bounding set in the first formula is another per thread attribute. It is used to limit capabilities handed over during execve. This set value is preserved across fork and execve and once something was dropped from it, it may not be regained back. Task may not add any capability not included in its bounding set to its inherited set and the bounding set also masks the file permitted set. Therefore only way to pass cabality not included in the bounding set is in case the inherited set contains this capability(added before shrinking capability bounding set) and executed binary has that capability in its inherited file capability set.

To keep original super user semantics, following corrections are done during execve: if an executed program belongs to root and has the uid bit set or the real uid of the thread is 0 (root) then all capabilities in the file permitted and inherited set are raised and also the effective bit is set. This grants all capabilities to that executed thread with the exception of those not included in the bounding set.

During the transitions between root and non-root uid of a running task, its capability sets are also changed respectively. The transition of effective uid from 0 to nonzero means clearing effective set, the transition from the state when there is some uid equal to 0 to the state when all uids are nonzero means clearing the permitted set. Finally the transition from nonzero to 0 copies whole permitted set to the effective set.

The capability policy also includes three per process bit flags modifying the behaviour for threads with uid 0. These are:

- SECURE_KEEP_CAPS - if set, switching all uids to nonzero does not mean losing permitted and effective capabilities, resets after each execve
- SECURE_NO_SETUID_FIXUP - changing uids of thread does not change capability sets

- SECURE_NOROOT - disables granting all file capabilities to set-uid-root programs and programs with uid 0

Finally there is also a companion locking flag for all these, which disables any future changes to these if set(and also changes of locks itself).

4.5.1 Necessary changes in capabilities policy

Now the question is whether it is possible to realize specially authorized trusted subjects using the mechanics above without any modifications. What we actually want is that only certain programs could gain CAP_MAC_OVERRIDE capability and no others. This can be achieved only by editing file capabilities of these programs with mentioned capability, such that they gain this capability in its permitted set and have their effective bit set. Attempt to grant such setting to any other programs should be denied. Privilege to setting file capabilities is maintained by CAP_SETFCAP capability. The only way to deny setting CAP_MAC_OVERRIDE for any process by root processes is to drop CAP_SETFCAP from the capability bounding set of init(or somewhere lower inside the process tree, where arbitrary root process may be executed). This however means disabling the possibility to assign any file capabilities within the system, what is an undesired behaviour. There is no way to specialize this restriction to only single CAP_MAC_OVERRIDE capability.

Fortunately capabilities policy may be overriden inside LSM hooks, it is possible within the *inode_setxattr* hook to test whether the XATTR_NAME_CAPS extended attibute(holds file capabilities) is set, and if yes look whether $CAP\_MAC\_OVERRIDE$ is being set and deny setting xattr in this case.

Another problem is computing new thread's capabilities during execve. We want only threads executed from a binary with the CAP_MAC_OVERRIDE file capability set to be able to gain the CAP_MAC_OVERRIDE effective capability regardless of the uid of executing thread. Let's assume security bit SECURE_NOROOT is not systemwide set. If it was, too many programs with uid 0 requiring privileges for their correct behaviour(other than CAP_MAC_OVERRIDE) would not work anymore, because although the file capabilities infrastructure is developed, it is not

extensively used within Linux distributions, which rely on keeping the superuser semantics during execve. Fine tuning of file capabilities for privileged programs by system administrator would be required.

But with SECURE_NOROOT turned off, every program executed by a root process gains CAP_MAC_OVERRIDE automatically if not masked out by the capability bounding set. In case it was masked out however, also running the intended SAT program would not grant him CAP_MAC_OVERRIDE.

The solution may be realized by the

```
int bprm_set_creds(struct linux_binprm *bprm)
```

LSM hook that is called prior to succesful execve operation on a binary represented by passed argument. This hook may affect the security credentials of a new task being run from this binary, which contains also its capability sets. It is possible to navigate to the respective dentry of a binary file from the *bprm* structure, check there whether its file capabilities contain CAP_MAC_OVERRIDE and assign this capability to the effective and permitted set of the executed task only in case yes. The behaviour of execve for all other capabilities will stay unchanged by calling the bprm_set_creds hook of the capability policy implementation within my hook.

4.5.2 Implementing specially authorized trusted subjects

Now when I have shown how it is possible to recognize specially authorized trusted subjects within the kernel, it is time to implement them as user space utilities.

First utility is called the extended attribute manager *xattrmng*. It is used for maintaining the security policy relevant xattr PTS on objects of a file system. It provides these main functions:

- printing the current security context of an object in the textual representation
- removing the PTS extended attribute - this means setting object's security context to the default state
- setting the PTS extended attribute to desired value

Remove and set operations are sensitive, before allowing them, the utility checks the equality of the uid of the running process with the changed object's owner uid. If this check is succesful the owner is asked to authenticate himself. Authentication is done by password. Here several implementation possibilities are offered.

The implemented policy may maintain its own security authenticating system for users independent of authentication into the system. This would entail password changes managment, storing hashed values of passwords and protection of stored passwords against modifying.

Another possibility is to use system passwords for authentication. These are stored in /etc/shadow. The file is formatted into rows, each row corresponds to a single user identified by its UID number. The most important column of a row is a hashed password. It consists of three parts, first identifies the algorithm used for hashing, the second is a random salt string concatenated with a raw password before it is hashed, and third is the actual hashed value.

Validating of the passed password could be done by the utility itself, this would however require the detailed knowledge of storing passwords and computing hashes, and could also become potentially outdated in the future.

I used Pluggable Authentication Modules libraries providing a high level API for using the lower authentication schemes in the background. Utilities such as *login* or *su* use PAM libraries for authentication tasks, therefore there is a high probability of keeping the same API and a given functionality without needed changes in the future. The authentication part in utilities execution starts PAM session with the passed system user(uid of the running utility) and communicating objects, then program calls the *pam_authenticate* service, which asks user to pass its password and finally the result of authentication is passed to the program. In case the authentication was succesful the actual remove/set xattr functionality is executed.

In case of the set operation, the passed security context value is validated and if valid it is passed via *setxattr* to the kernel. In the policy hook, the CAP_MAC_OVERRIDE capability of a process requiring change is checked and afterwards the validating passed value is done as well.

Except from this remove/set PTS xattr functionality done on a single object I

have implemented also possibility to do it recursively in a passed directory to ease the mass labeling of objects of the same security sensitivity.
Examples of *xattr* utility usage:

```
xattrmng -g /etc/shadow
xattrmng -d ./text
xattrmng -s "c_o=2;i_o=2;" /etc/shadow
xattrmng -sr  "c_o=0;i_o=2;" /bin
```

The first example reads the security attributes of /etc/shadow, the second removes the PTS xattr from the text file, the third sets the security attributes of /etc/shadow with the passed value and the fourth uses recursive setting on /bin directory.

The second SAT utility is called *procmng*. This utility enables user to run a process with desired security attributes. First it asks the user to authenticate itself using PAM and afterwards it validates the desired security attributes and writes them to /proc/self/attr/current. This setting is evaluated within *setprocattr* LSM hook, it checkes whether current process has the CAP_MAC_OVERRIDE capability and validates passed value. If everything is alright, the *procmng* process now runs with the passed security attributes. Afterwards it executes the program desired by the user and since execve preserves security attributes, the required task is finished and program runs with the security attributes required. To easy up repetitive running programs with commonly used settings to users, I enabled passing security attributes' value as a link to file, which content is read and used. So the user may store commonly used configurations in security profile files.
Example of reading the protected file /etc/shadow with cat program, security attributes are passed either directly or by a profile file:

```
procmng "cr_s=2;" cat /etc/shadow
procmng -p PrivilegedRead cat /etc/shadow
```

The correct setting of the enviroment for these utilities prior to the policy deployment includes setting the CAP_MAC_OVERRIDE file capability for their binaries.

4.6 Necessary exceptions to running trusted subjects

At this stage, when the core of security policy is implemented, it is time to test it on the running system with proper binding of security attributes to objects according their sensitivity. I will list initial labeling done to important permanent objects used in the system and try to clarify chosen labeling. Security attributes will be expressed by the textual representation.

- system libraries and binary executable files $c_o = 0; i_o = 2;$ - these should be readable by every subject, information they hold is public and needed for running tasks, however they have to be strongly protected against unauthorized modification, which could cause unpredictable or malicious behaviour of running tasks(included directories and their contents - /bin, /lib, /sbin, /usr/bin, /usr/sbin, /usr/lib, ...)

- system configuration files - $i_o = 2;$ their unauthorized modification is strictly unwanted, affecting the system behaviour, confidentiality level is not completely clear, for most files default level is sufficient, might be derived from the DAC access rules, if others do not have read permission then confidentiality level should be 2 (for example /etc/shadow), configuration files are included mainly in /etc directory

Now I have tried to test system with these permanent objects set. Booting system was not succesful with kernel panic as a result, error messages stated that main harddisk could not be mounted. At this occasion, it is suitable to emphasize the importance of having standard kernel available for the system loading at boot menu to be able to enter system and try to fix things in tested version. Apparently, what happened was that some process running with default security attributes during boot up sequence tried to execute operation on protected object, which was rejected by loaded PTS policy. In order to recognize effectively participating entities, denied operation and other circumstances and be able to look for solutions I introduced permissive mode of running security policy.

4.6.1 Permissive mode

As a response to above described problems I added two modes of running into the security policy. They are independent of each other and affect policy behaviour in different way.

Printing mode records all events when policy decides to reject privileges to perform some operation with respective details. That means printing rejected operations, identifiers of entities concerned in operators(for inode it is inode number and type of filesystem, for dentry it may be recursive printing of its path, for process it may be its name) and maybe printing important security attributes of these entities.

Permissive mode is used for tuning system security. Running permissive mode without printing mode does not make any sense. During permissive mode, all operations are judged by the policy in standard manners, but in case operation would be denied, print is made(printing mode on) and the hook does not return -EPERM but permits operation. This mode therefore only simulates implemented policy and logs possible problems, but from the outer view it seems as if no mandatory policy would be loaded. To see kernel live time logs *dmesg* command may be used.

Printing and permissive modes are realized in all access control hooks, they may be set on or off by defining respective macro values. To easily switch between them without recompiling kernel, I have implemented two policy variants with distinctive security operations tables and names, one implementing permissive mode with prints, second having nonpermissive mode with prints. Which policy will be loaded is chosen by setting security boot parameter.

4.6.2 Default binary execution attributes

Thanks to permissive mode I was able to detect that mount process running with default security context tried to create file within /etc directory protected against write and was rejected. However mount process was executed during boot up session prior to user has any control over system, therefore is not possible to authenticate itself and execute mount with proper attributes.

There must be found the way how to grant needed privileges to such processes.

First idea is to divide system run into two epochs: booting and the state when user is in control of the system. Processes executed during booting would be able to override all access control checks(running as trusted subjects). Several problems raise for this approach. How to reliably differ between these time periods? Another problem is that after the start of the second epoch there are still running processes executed during boot. If we could reset their security attributes to default values automatically, this could bring unwanted effects.

The rational solution seems to find as tight ranges of security attributes which they need to operate within as possible and grant them these security attributes automatically. This is naturally realized by binding these default security contexts to the respective binaries, from which these processes are executed. Binding may be done by using extended attributes.
For this purpose security policy reserves new security attribute BIN_PTS by following macros:

```
#define XATTR_BIN_PTS_SUFFIX "BIN_PTS"
#define XATTR_NAME_BIN_PTS XATTR_SECURITY_PREFIX XATTR_BIN_PTS_SUFFIX
```

Now the question is how to manage setting this attribute to prevent violating policy. If malicious process could set this xattr, it would be able to run arbitrary program with arbitrary security context. Therefore utility for setting BIN_PTS xattr representing default security context of process executed from binary should run as specially authorized trusted subject. To recognize it within kernel, CAP_MAC_OVERRIDE file capability would be assigned to it similarly to previously implemented SAT subjects.

Another question is whose authentication is required within that utility and under what conditions will process being run from binary gain security context described by the value of its BIN_PTS xattribute value. We could require at least the authentication of the binary's owner during setting the xattr, or to be more strict only security administrator could do such changes. Both of the solutions are suitable. The first answer partially implies the second question's solution. Receiving binary's default security context during executing should be granted to processes whose owner was authorized to set this security context for binary. In this case,

it is the owner of the binary(in case of setuid binaries it should be every process executing it). Since programs being run during boot that need default security context are probably all owned by root and bootup processes have uid of root, this solution seems viable.

To implement this behaviour, *bprm_set_creds* hook is modified. In case the currently running process has uid equal to the owner of executed binary or binary is setuid, it is checked whether binary has BIN_PTS xattr set with valid value. If yes, the process being executed does not inherit security context of current process, but its security context is read from the binary xattr.

With adding the concept of default binary execution attributes another issue has come out. When the security context of process is gained from BIN_PTS xattr and afterwards the process execve's another process(without BIN_PTS xattr), what should be security attributes of the new process? This problem shows up practically, because login manager needs to access system passwords, therefore needs cr_s=2. However after succesful login all processes of logged user automatically gain cr_s=2 by applying traditional transfer of security attributes between processes. It is clear that in this case, it is unwanted.

To solve this problem I have introduced the new security attribute *heritable* for subjects(into subject_pts structure). It has integer value and tells whether the security attributes should be preserved through execve operation and how many times. It recognizes positive values with zero on one hand and infinity value on the other hand, which is modelled by -1 value. During execve, this value is checked, if it is -1 then new process gains the same security context as the old one. If value is positive, then the same happens, but heritable value is decreased. If value is equal to 0, then new process does not inherit security attributes of the old one, but it gains default security attributes.

If the binary has BIN_PTS xattr set and its value says nothing about heritable attribute, than heritable is supposed to be 0. In case the process with default security attributes is created or process launched by user using *procmng* utility and heritable is not explicitly set, then heritable value is expected to be -1.

Finally I will describe the userspace utility for maintaining BIN_PTS xattrs on

binaries. It is called *binmng* and it supports reading current value of xattr, removing value, or setting it with explicitly passed value or value read from security profile file.
Examples of use, first one is reading xattr, second is setting the xattr:

```
binmng -g /bin/mount
binmng -s "iw_s=2;" /bin/mount
```

This utility requires to have CAP_MAC_OVERRIDE file capability.

4.7 Labeling temporary objects

In this section I will focus on dealing with the security attributes of nonpermanent filesystem objects. These objects are dynamically created usually during the system startup. I have already mentioned how it is possible to label them uniformly within the *d_instantiate* hook according to filesystem they belong to. This approach is however not always sufficient. I will focus on device filesystem objects, which contain representatives that require a special security treatment.
Here are some of them with the description of suitable security attributes for them:

- device files representing hard disks and partitions, by accessing them, process may read or modify any data psychically found on the device, this is a very powerful privilege, it can be seen that for every process running on some arbitrary security level there exists an object with some security attributes which process cannot read/write to, let's assume a process may read/write to the particular device file, then however this device might contain an object for which the process has access denied but could bypass this by accessing the object directly through the device file, this implies that process with no conventional security context should be able to read from/write to these devices

- special devices such as /dev/null (everything written to it is accepted, reading returns EOF), /dev/zero (for read returns as many zeroes as asked), /dev/full (write returns No space left error), every write and read operation on these

devices should be granted, however if we would assign these devices some particular security attributes, there would always exist security attributes for process, that would forbid read/write

- /dev/random , /dev/urandom - these act as random numbers generators, read access should be granted for everyone

4.7.1 Security attributes outside standard bounds

These examples show up that so far used security attributes are not sufficient for some devices. Therefore we have to widen the security model. One possibility is to add standalone flags for each combination of confindentiality/integrity and access for everybody/nobody. This is however reduntant and makes model less clear. It is possible to use current attributes with the values not interpreted so far.

This means that setting the value of object's security attribute above the upper bound is restrictive towards all operations on the object considering this security attribute. On the other hand, values below the lower bound grant access to everybody. In our enviroment this means for example setting $c\_o = -1; i\_o = -1;$ to /dev/null and $c\_o = 3; i\_o = 3;$ to /dev/sda1.

Now it is important to think of, who should be authorized to set the attributes outside normal bounds on the object. If we would permit setting arbitrary security value to the object by its owner, he would be able to significantly affect behaviour of other users' processes. Let's assume there is a process P1 of user U1 running with the $ir\_s = 2;$ security context. This process shouldn't be permitted to read from objects of another user. But if another user U2 could set $c\_o = -1;$ on its objects, this would mean that P1 is allowed to read such objects. Therefore setting security attributes should be allowed only to the security administrator. Details about the security administrator role and security administration mode are in the last section of this chapter. Labeling of these specially treated devices has to be therefore done by special trusted subjects authorized by the security administrator.

4.7.2 Udev daemon labeling

Another question is how the actual labeling of devices will be done. Simple solution is to run a script labeling devices in the moment when they are already created and inserted into the directory tree. This however let them with the default security context in the time between their initialization and running script. Another problem is labeling devices that are created later during system run, for example when plugging external disc. More elegant solution is to use the *udev* daemon. Udev is a user space process managing device filesystem standardly mounted at /dev. It reacts to events triggered during plugging and loading devices within kernel by reading the sysfs filesystem mounted at /sys through which kernel exports information about various curretly plugged devices. As a response to these events it creates corresponding device nodes within the device filesystem.

Udev's behaviour is managed by the rules found in udev's configuration files. [12] Each rule consists of a series of key-value pairs. There are pairs that specify upon which event this rule should be used, the specification might be done by the kernel name of the device, action done, or various attributes of the device. Specification may be used for wide set of devices having common attributes because it may be expressed also using regular expressions. Then there are pairs related to making device nodes specifying the name of created nodes or alternative nodes being symbolic linkes to the real node. The most interesting pair for us are those that enable running user specified command upon recognized event specified by this rule.

The labeling process run by udev when it recognizes specially labeled device has to be special trusted subject authorized by the security administrator because of setting attributes outside traditional bounds. Let's assume we may mark particular binary, so that when the process is run from it, this process is recognized as a special trusted subject authorized by security administrator within kernel and it allows the process labeling objects with arbitrary values. I will show how this marking may be done in the next section. Since this labeling process run from the marked binary has to provide labeling automatically without possibility to authenticate as the security administrator, it is crucial that no process apart from udev should be able to run this

binary. This may be realized also with the security attributes raised above standard upper bound. If security administrator sets security context of the device labeling utility binary to $c\_o = 3$; no process with standard attributes will be able to read it and therefore also not able to execute it and subsequently change arbitrary object's security attributes. To enable udev running this utility, the security administrator has to set the BIN_PTS xattr of udev to $cr\_s = 3$; so that it will be able to execute it. This will cause that udev will be run automatically with the raised read capability during boot up.

The mentioned utility was named *deviceXattrmng*. It has the same functionality as *xattrmng*, but it does not require authentication.

To prevent misusing of potential bugs found in udev by attacker for reading other objects with a high confidentiality protection we may use the partially trusted subjects concept. The *deviceXattrmng* binary may be assigned a label l_o apart from only raised c_o and the BIN_PTS xattr of udev will get only $crl\_s = 3$; and $crls\_s$ set will have the respective label added.

4.8 Security administration

It is clear from the previous section that there is a high demand on adopting a security administrator role for the implemented security model.

The first important question is who should serve this role. In traditional Linux, the root account gains all administrative privileges, this means he maintains configuration of the system, network, security etc. Not only the aggregation of so many privileges in hands of a single person is problematic, but also the fact that all processes running in the system on behalf of root have the full power. It is therefore reasonable to separate the role of the security administrator of our policy from the system administrator root account. This does not necessary mean that the traditional system administator and the security administrator have to be two distinct real persons, but it enables such configuration of privileges.

Another problem is the extent of the security administrator privileges granted by the security policy. Natural solution enables the security administrator to override

completely all policy access controls and rules. These privileges granted by kernel to the security administrator are however strongly influenced by the form of recognized security administration mode. If we realized security administration by creating a system account for the security administrator and stated that all processes running with the effective uid of this account gain security administration privileges, then in case the system administrator could log into his account he would gain unlimited privileges. On the other hand, in the same situation but with the logging disabled for the security administrator account, the administration mode would be restricted by the functionality of utilities belonging to security administrator and having the setuid bit turned on.

I have used capabilities for recognizing security administration. Processes holding the CAP_MAC_ADMIN capability are privileged to do security administration of the policy.The ideal solution stating clearly authority of the security administrator is creating a set of utilities with well defined semantics that run as trusted subjects authorized by the security administrator and only they execute security administration operations. The solution how to grant the CAP_MAC_ADMIN capability only to these utilities is similar to the realization of SAT subjects for managing security attributes by the entity owner. I have modified capability transfer during execve for CAP_MAC_ADMIN so, that only processes executed from binaries having the file capability CAP_MAC_ADMIN may enforce the administration of security. This concerns the beginning of the administration.

In order to enable powerful utilities starting the administration session (for example running bash with the CAP_MAC_ADMIN capability) through which all executed programs should have the administration privilege, I have further modified the capability settings within *bprm_set_creds* hook. Executed process gains the MAC administration capability also in case the old process making exec had this capability.

Another issue with the trusted subjects running administration tasks is how they authenticate the security administrator. I have used authentication by password. The important thing is the way of storing, validating and change managment of this password(and all other passwords used for the authentication within SAT subjects

recognized by the policy). One possibility is to store an independent set of passwords for the users(not connected with their system accounts). This way I would have to implement utilities for managment of these passwords, and would bring an additional overhead for the users to memorize and manage two passwords. The better solution is using the system accounts for authentication, also taking into an account the use of PAM authentication(validates passwords according to /etc/shadow) in the already implemented utilities. After this decision, it is necessary to create account for the security administrator before the policy deployment.

In my case, I have created the user *securityAdministrator* with the uid number 999. Numbers lower than 1000 usually determines a special system user. Number 999 should be free in most used Linux distributions(after the uid 0 of root there are always numbers reserved for various virtual users). The implemented utilities for security administration need to know whose authentication they will ask for. The uid might be hardcoded into them or another viable solution is to create a configuration file (for example in /etc) strictly protected against modification and readable by everyone, which will contain uid of the security administrator.

Finally I will mention some utilities suitable for the security administration and their functionality. It depends on a decision before the deployment of the security policy which utilities to provide to the security administrator by assigning CAP_MAC_ADMIN to their file capabilities. In less dynamic systems, where not so many changes from a security point of view are done during the production run it is suitable to limit privileges of the security administrator and rely on the correct initial configuration. On the other hand, in dynamic systems, security administrator may have very high privileges. In this case security might be raised by requiring more than one administrator for running security administration utilities. This may be implemented by easy modification of utilities, so that the configuration file identifying security administrator would contain the list of uid numbers, and the utility would continously ask authentication from particular security administrators in this list. Administration mode will start only in case all of them succesfully log in.

Tips for functionality provided by possible security administration utilities:

- labeling device files - implemented *deviceXattrmng*, called by udev, not re-

quiring authentication of the security administrator, therefore having confidentialy level above the standard upper bound

- labeling passed object with arbitrary attributes(also outside the normal bounds, such object afterwards is protected against relabelling by its owner as well), implemented *adminXattrmng* utility
- labelling passed binary with the default execution attributes - these may exceed standard ranges, used for programs working with special objects, implemented *adminBinmng* utility
- maintaining user passwords, may change other users' passwords, implemented *pswd* utility
- setting CAP_MAC_ADMIN and/or CAP_MAC_OVERRIDE on binaries, adding new specially authorized trusted subjects
- running administrative session - all processes run with the CAP_MAC_ADMIN capability, the last two have the same power, the security administrator is high privileged, implemented *admin* utility
- changing the owner of the object
- setting the CAP_CHOWN, CAP_SETUID file capabilities - enabling process to change its effective uid or change owner of the objects, these operations will be discussed in the next subsection

4.8.1 Restricting security administration privileges of root

This part is going to discuss some necessary steps needed for limiting root privileges concerning possible policy violation by him. This is done as a result of the decision to separate the role of system administrator from the security administrator.

The main idea of this process is to prevent root from doing such operations, which have been granted to him so far and which should be done only by the security administrator(these do not necessary have to be straight operation requests, but

also hidden channels or indirect operations such as directly accessing devices). The detailed knowledge of the system is required to do this properly, I will mention only important issues known to me, certainly there are many more. It is important to realize, that system administrator has power to bring down the system or corrupt it in many ways. The idea is not to control these types of misbehaviour including accessibility violation, but only restrict violating of the implemented security model.

System accounts managment

Traditionally system accounts information are stored within /etc/passwd and /etc/shadow (storing hashed passwords). Utilities such as *passwd* are used to manage these information. It is a setuid program allowing each user to change its password etc. and root to make arbitrary changes. Since passwords for system accounts are used in our policy for authenticating users and especially the system administrator, these should be protected against modification by anybody(included root) except from the security administrator. Depending on the chosen administration policy, system administrator does not have to be allowed to change it directly but with use of trusted administration utilities.

The recommended setting of the security attributes of /etc/shadow includes raised confidentiality and integrity. Since this file is accessed also by programs such as login managers, which run with the root uid number, root should stay its owner. This way the non-privileged processes of root won't be able to access it and also no processes of other users. The read access of login managers may be solved using default execution parameters for their binaries. However we want to completely prevent root processes from modifying this file. This may be achieved by raising the integrity level of /etc/shadow above the standard bound to value 3. Now the standard passwd utility run by root with arbitrary security values won't be able to modify it. While trying to modify /etc/shadow, passwd works in two steps. First it makes a copy of original file, then it tries to do required changes on this copy and finally if all went right it calls move from the changed copy to the original file. Since the kernel hook finds out that the dentry to which move is made is already engaged, it verifies write permission to it and this denies modification of the password.

To enable such changes to the security administrator I have implemented the *pswd* utility. It runs with the CAP_SET_UID capability enabling changes of own uid and also with the CAP_MAC_ADMIN privilege, therefore overrides access control on /etc/shadow. It could either try to do required changes itself, but this is too laborious and prone to errors, therefore uses the proved *passwd* program and is just a wrapper for its execution.
I will describe behaviour of this utility during the request for a password change. The utility may be executed with no argument, this means user wants to change own password. In this case *passwd* is executed directly, it knows when there is no argument, the user wants to change its own password. It may modify /etc/shadow now because the CAP_MAC_ADMIN capability is inherited.
In case *pswd* is called with a argument, it supposes that the security administrator wants to change the password of passed user. Therefore it challenges the security administrator to authentify itself. If succesful, the running process changes its effective and real uid to 0 and calls *passwd* with the passed user. Changing uids to root is important, because passwd enables changing passwords only to root.

Maintaining software and libraries

The system administrator is responsible for maintenance of software in the computer. By modifying or replacing installed software or libraries by malicious ones violation of the policy may happen. Modifying specially authorized trusted subjects enforcing the policy is even more dangerous. Therefore already installed software and libraries have to be protected against unauthorized modification by the system administrator by limiting provided ways of maintenance.

This could be achieved by raising the integrity level of the directories containing programs and libraries (and their content) to 3. In addition, software maintenance will be restricted only to usage of a trusted software package installer(or its wrapper), which could be for example limited to installing software from the official repositaries of a distribution. This program would have the default execution attributes set to $iw\_s = 3; cw\_s = 0; cn\_s = 0; in\_s = 3;$, so it would be able to modify the contents of installation directories and ensure that newly created files have proper

security attributes. It would ask root for authentication after start.

Another possibility how to implement this similarly would be changing the owner of installation directories and its contents to a virtual user created for software maintenance. This user would not be able to log into the system and since all software would have increased integrity, other users' processes could not modify it. The software would be maintained through a setuid installation utility belonging to the virtual user(with raised default execution attributes) and could have the same functionality as one described paragraph above.

Changing ownernship of objects

This is one of the crucial operations I have not paid attention to yet. It should be reserved only for the security administrator or forbidden at all. I have analyzed the possible impact of forbidding it with the security policy running in a permissive mode with prints. The vast majority of chown operations were done during the system startup or after the user logged in. I have watched the transitions between users, most have been done on temporary objects and changing ownership between root and one of the virtual users. Only process changing ownership between root and my account was the login manager after I logged in.
From these observations I have decided to forbid chown operations with the following exceptions:

- changing ownership between root and virtual users on objects is permitted in case the process requiring chown is CAP_CHOWN capable, virtual users are used usually to enhance the security of important system objects through group permissions in DAC, this should not represent a security threat in our model

- other changes between the real users have to be allowed at least to some extent, for example exception for login managers, trusted subjects privileged to call chown should be recognized by marking the binaries they are executed from, this marking is allowed only by the security administrator and is done by setting the CAP_CHOWN file capability to it, remembering whether thread is

privileged to make such exceptions cannot be held within the CAP_CHOWN thread capability because this is already used as a condition controlled wihin the first type of exceptions, therefore I will not modify mechanism of computing this thread capability and will remember the privilege to execute chown according to this second exception as a new boolean member *chown* stored in the *subject_pts* structure, this member is initialized within the *bprm_set_creds* hook according to presence of the CAP_CHOWN file capability in executed binary

Important question is how to distinguish users concerned in the first exception. Their list might be communicated to kernel by trusted subject during boot up. Much easier solution I used and which is sufficient in case the list of virtual users is stable after the initial configuration of the system is a firm setting of the array of user identifiers *system_users* in the kernel patch. This array includes root and all virtual system users.

Changing process uid number

This operation is critical within our security model. If user would be able to run a process with the raised attributes which would afterwards change own uid number, the process could access protected data of another user. I have analyzed the running system and logged attempts to change the process uid number. The results are very similar to the chown operation. Except from the login manager, all uid changes are done between the root and the virtual users. I have therefore used a similar solution as for the chown operation recognizing two types of exceptions. The second type is approved only to the programs executed from binaries having the file capability CAP_SETUID set. Setting these may be done only by the security administrator. To keep track of this privilege within a running thread, there has beed added a *setuid* member to the *subject_pts* structure being set in the *bprm_set_creds* hook. The list of *system_users* used to recognize , whether the change uid operation is classified as a first exception, is the same and shared with the chown operation.

Chapter 5

Testing

In this chapter, I will show a guide for installation of the implemented policy and user space utilities and example of their usage.

5.1 Kernel patch installation

This section includes the tutorial for applying a kernel patch with the implemented policy.

1. First, you need to download Linux kernel source code, I recommend the version you are currently using. You may find this by running $uname - r$ command.

2. Navigate to the root directory of the downloaded source. Add following line to the $./security/Makefile$

   ```
   obj-y  += pts/built-in.o
   ```

3. Copy the $/pts$ directory from the enclosed archive to $/security$ directory of the kernel source tree

4. Check configuration before compiling, make sure extended attributes support is enabled. I recommend using a copy of the cofiguration file for currently running kernel. In my case it is $/boot/config - 2.6.32 - 28 - generic$

5. Build and install the patched kernel. For Ubuntu 10.04 this manual can be used. [13] Now check whether an entry for your newly built kernel exists in your boot loader. In my case these entries can be found at $/boot/grub/grub.cfg$ .I recommend keeping also possibility to load the original kernel and make two entries for the new kernel, one running in permissive mode. Add boot options $security = pts$ to the entry for the new kernel and $security = ptsPermissive$ for the entry loading kernel with policy in the permissive mode.

6. Make sure extended attributes are enabled for mounted hard drives. Add the $user\_xattr$ mount option for them in the $/etc/mtab$.
   The line for my hard drive looks like this:

   ```
   /dev/sda6 / ext4 rw,errors=remount-ro,user_xattr 0 0
   ```

5.2 User space utilities installation and configuring security

This section includes a guide for installing user space utilities for managing the policy and recommends security attributes setting for objects of the file system and default execution options for some programs.

1. In order to compile user space utilities, development libraries for extended attributes and Pluggable Authentication Modules are needed. For Ubuntu distribution these are included in $libattr1-dev$ and $libpam0g-dev$ packages.

2. To install user space utilities run $sh\ installUtilities.sh$ script from the enclosed archive as a system administrator. The script builds them, moves to /bin directory and sets required file capabilities.

3. create security administrator account

Now it is time to bind the security attributes to important objects. I will mention only some recommended settings.

- device labeling - copy enclosed configuration file $40 - device - xattrs.rules$ for udev to the */etc/udev/rules.d* directory, edit for your specific devices if needed, raise integrity level of the directory and its content to 3
- label shared libraries and programs with confidentiality 0, integrity 3, example $adminXattrmng\ -sr\ "c\_o = 0; i\_o = 3;"\ /bin$
- label contents of /etc with integrity 2, the exceptions are /etc/shadow (C2,I3) and /etc/passwd(C1,I3)
- protect /boot and its content from modification by anybody except security administrator by raising I to 3
- try to run the policy in the permissive mode, watch kernel logs made during boot up, set default execution parameters using *binmng* and *adminBinmng* utility for trusted programs accessing protected objects during booting, in my case these were $udev$, $mount$, $NetworkManager$, $gdm-binary$, $unix\_chkpwd$ ...

5.3 Running program with user specified security attributes

This section describes an example of running program with nondefault security attributes. It is a program downloaded from the internet, that is untrustworthy, but we want to test out what it does. We run it with lowered security attributes so that it cannot read from/write to files that are on normal or higher security level.

```
procmng "cr_s=0;iw_s=0;" ./unknownApp
```

Let's say the program is malicious and tries to delete entire home directory of the user. However, the security module does not grant it the privileges to do so. If it was run with the default security attributes, it would cause great harm to the user.

Conclusions

The objective of this thesis was implementing Two-Dimensional Labelled Security Model with Partially Trusted Subjects in Linux operating system. We have shown the feasebility of implementation by creating the prototype model using Linux Security Modules architecture and necessary user space tools for the policy managment. The implemented model in its current state is focusing mainly on the protection of filesystem objects. It introduces the role of security administrator for the policy maintenance. The thesis presents several concrete examples of running applications within the implemented enviroment and gives the hints for correct security configuration of the system.

For production deployment of the security model without any exception in constrain rules, the following tasks need to be done in the future:

1. Communication objects protection - implementing security attributes and rules for the network communication subjects

2. Trusted path - forbidding information flow between processes via the X server by using separate X server for each specific combination of security attributes of subjects

Bibliography

[1] Janáček Jaroslav. *General Purpose Operating System for Security-Critical Applications*. PhD thesis, Univerzita Komenského, Bratislava, 2010.

[2] Matthew Fillpot. Understanding linux file permissions. 2010. `http://www.linux.com/learn/tutorials/309527-understanding-linux-file-permissions`.

[3] Andreas Gruenbacher. POSIX access control lists on linux. 2003. `http://www.suse.de/~agruen/acl/linux-acls/online/`.

[4] Mandatory access control. `http://www.cis.syr.edu/~wedu/Teaching/cis643/LectureNotes_New/MAC.pdf`.

[5] SELinux documentation. `http://selinuxproject.org`.

[6] Wright CH., Cowan C., Smalley S., Morris J., Kroah-Hartman G. Linux security modules: General security support for the linux kernel. 2002. `http://www.usenix.org/event/sec02/full_papers/wright/wright.pdf`.

[7] Wright CH., Cowan C., Smalley S., Morris J., Kroah-Hartman G. Linux security module framework. `http://www.kroah.com/linux/talks/ols_2002_lsm_paper/lsm.pdf`.

[8] Jurčík M. Using selinux to enforce two-dimensional labelled security model with partially trusted subjects, 2010.

http://www.dcs.fmph.uniba.sk/bakalarky/obhajene/getfile.php/bc_jurcik.pdf?id=133&fid=254&type=application/pdf.

[9] Jake Edge. LSM: loadable or static? 2007. http://lwn.net/Articles/255650/.

[10] LXR - the Linux Cross Reference. http://lxr.linux.no/.

[11] Capabilities, Linux Programmer's Manual. 2010. http://www.kernel.org/doc/man-pages/online/pages/man7/capabilities.7.html.

[12] Daniel Drake. Writing udev rules. 2006. http://reactivated.net/writing_udev_rules.html.

[13] Ubuntu KernelCompile documentation. https://help.ubuntu.com/community/Kernel/Compile.

[14] Ori Pomerantz Peter Jay Salzman, Michael Burian. The linux kernel module programming guide. 2007. http://tldp.org/LDP/lkmpg/2.6/html/lkmpg.html.

[15] Robert Love. *Linux Kernel Development*. Novell Press; 2 edition, 2005.

Resumé

V tejto práci sa zaoberáme implementáciou "Two-Dimensional Labelled Security Model with Partially Trusted Subjects" bezpečnostného modelu v operačnom systéme Linux. Model navrhol RNDr. Jaroslav Janáček, PhD. vo svojej dizertačnej práci.[1] Bezpečnostná politika sa zameriava na ochranu dôvernosti a integrity citlivej informácie spracovávanej privilegovanou aplikáciou pred nedôveryhodnými aplikáciami toho istého používateľa. Takúto ochranu informácie nie je možné realizovať štandardnými metódami voľného riadenia prístupu v operačnom systéme. Implementácia má formu patchu do jadra systému a sady podporných nástrojov do užívateľského priestoru umožňujúcich administráciu a vykonávanie autorizovaných výnimiek z pravidiel bezpečnostnej politiky.

Cieľom je ukázať uskutočniteľnosť realizácie politiky s využitím bezpečnostnej architektúry Linux Security Modules v jadre systému. Architektúra poskytuje všeobecné možnosti realizácie povinného riadenia prístupu. Zaregistrovanej bezpečnostnej politike umožňuje pomocou tzv. "hook" funkcií sprostredkovávať prístup ku chráneným štruktúram jadra systému počas spracovávania rôznych systémových volaní dôležitých z hľadiska bezpečnosti. Navyše architektúra pridala ku chráneným štruktúram bezpečnostné polia umožňujúce politike ukladať pre ne ľubovoľné bezpečnostné kontexty.

Práca uskutočňuje mapovanie abstraktných entít v modeli na reálne štruktúry v jadre systému a taktiež mapovanie abstraktných operácii na jednotlivé kontrolné funkcie realizovaného modelu. Bezpečnostné atribúty trvalých objektov sú uchovávané pomocou rozšírených atribútov súborového systému. Implementácia využíva v jadre existujúci koncept oprávnení "POSIX Capabilities" na implementáciu špeciál-

nych autorizavaných dôveryhodných subjektov pod priamou kontrolou uživateľa systému. Tieto subjekty umožňujú uživateľovi po úspešnej autentifikácii meniť bezpečnostné koncepty vlastnených objektov alebo bežiacich subjektov. Na korektnú realizáciu politiky toku dát implementovaný model upravuje sémantiku výpočtu získaných používaných oprávnení pri spúšťaní nových subjektov.

Ďalej sa v práci rieši bezpečný spôsob spúšťania subjektov s uživateľom preddefinovanými bezpečnostnými atribútmi bez nutnosti priamej interakcie s uživateľom a automatické značkovanie dynamických objektov pri ich vzniku. Pri týchto riešeniach sa ukazuje potreba jemného rozšírenia hodnôt bezpečnostných atribútov kvôli špecifickým bezpečnostným požiadavkam niektorých existujúcich objektov. Rozoberá sa dopad týchto rozšírení na možnosti správy politiky.

Nakoniec sa rieši otázka nedôveryhodného systémového administrátora a zavádza sa rola bezpečnostného administrátora spravujúceho politiku s príslušnými implementáciami potrebných nástrojov. Práca ďalej pojednáva o tom, ako je možné jednoducho definovať privilégia bezpečnostného administrátora na základe poskytnutých nástrojov a realizuje potrebné kroky na odstránenie bezpečnostných privilégii systémového administrátora. Nakoniec je priložený manuál ku inštalácii realizovaného bezpečnostného modelu a odporúčané iniciálne nastavenia.

Výsledkom práce je prototyp bezpečnostného modelu zamieravajúceho sa hlavne na ochranu objektov súborového systému a sada nástrojov na správu politiky modelu. Na produkčné nasadenie politiky je nutné dopracovať niektoré úlohy popísane v závere.

Kľúčové slová: politika toku dát, bezpečnostný model, Linux Security Modules, POSIX capabilities