FEE CTU Prague
Department of Computer Science and Engineering

MASTER THESIS

Generating security description of applications for security frameworks

Pavol Lupták

Supervisor: Ing. Marek Zelem

Čestné prohlášení

I would like to thank Marek Zelem for his valuable suggestions, Andrei Badea and Martin Očkaják for their useful comments and corrections.

Prague, May 21th 2004

Pavol Lupták

Table of Contents

List of Figures

List of Tables

List of Examples

Annotation

The thesis introduces a new way of general-purpose representation of RBAC and DTE security models and analyses possible methods of their transformation. The first chapter evaluates the current state of the Linux kernel security considering existing security projects and describes a need for a general-purpose description of applications. The second chapter briefly describes well-known computer-security models and their existing Linux implementations. The third chapter focuses on the existing security models grammars in various security projects. The fourth chapter continuously extends the previous chapter and tries to outline a proposed XML grammar and its properties needed for RBAC and DTE representation. The fifth chapter considers about possible ways of transforming from the XML representation to SELinux and Medusa policies. The appendix chapter briefly outlines a proposed State-flow security model extension (SFSME), which significantly improves existing stateless security models. In the conclusion a future practical application of the XML security description is considered.

Anotácia

Diplomová práca predstavuje nový spôsob univerzálnej reprezentácie bezpečnostných modelov RBAC a DTE a analyzuje možné metódy ich transformácie. V prvej kapitole je zhodnotenie momentálnej situácie bezpečnosti v Linuxovom jadre uvažujúc pritom existujúce bezpečnostné projekty. Súčasne je popísaná potreba univerzálneho popisu aplikácií z hľadiska bezpečnosti. Druhá kapitola stručne popisuje známe bezpečnostné modely a ich existujúce Linuxové implementácie. Tretia kapitola sa zamerala na existujúce gramatiky bezpečnostných modelov v rôznych existujúcich bezpečnostných implementáciach. Štvrtá kapitola plynule naväzuje na predchádzajúcu kapitolu a pokúša sa načrtnút základné rysy navrhovanej XML gramatiky reprezentujúcej RBAC a DTE modely. Piata kapitola sa zamýšľa nad možnými spôsobmi transformácie XML reprezentácie na bezpečnostnú politiku SELinuxu a Medusy. V prílohe je stručne popísané navrhnuté rozšírenie o stavové prechody bezpečnostného modelu (SFSME), ktorý môže významne zlepšit existujúce bezstavové bezpečnostné modely. A nakoniec, v závere sú načrtnuté možné praktické aplikácie XML bezpečnostného popisu v budúcnosti.

Glossary

Due to terminology and interpretation inconsistency in the articles about security models and implementations, following uniform terminology for the fundamental technical terms has been used.

Discretionary Access Control (DAC)	A means of restricting access to objects based on the identity and need-to-know of the users and/or groups to which the object belongs. Controls are discretionary in the sense that a subject with a certain access permission is capable of passing that permission (directly or indirectly) to any other subject [DACdefinition]. The Unix DAC implementation uses an access control mechanism that enables the owner of a file or directory to grant or deny access to other users. The owner assigns read, write, and execute permissions to the owner, the user group to which the file belongs, and a category called other, which refers to all other unspecified users. The owner can also specify an access control list, which lets the owner assign permissions specifically to additional users and groups. Contrast with mandatory access control. See Also Mandatory Access Control.
Domain	A separate area for a running process denoted as *virtual space (VS)* in the Medusa DS9 and *type with domain attribute* in the SELinux. See Also Virtual Space, Type.
Domain and Type Enforcement (DTE)	DTE is a mandatory access control which assigns types to files and domains to processes. Access from domain to other domains/types is enforced according to the DTE policy See Also Mandatory Access Control, Mandatory Access Control, Mandatory Access Control.
Event	An event stands for any access of a subject to an object. The event is used in the context of a State flow security model extension (SFSME). See Also Transition event.
Flask Security Architecture (Flask)	Flask is an operating system security architecture that provides flexible support for mandatory access control model policies. SELinux is an implementation of this architecture.
Linux Security Modules (LSM)	The LSM project provides a lightweight, general purpose framework for access control. Access control models can be implemented as loadable kernel modules (e.g. SELinux, DTE for Linux)
Mandatory Access Control (MAC)	A mechanism for enforcing the corporate policy or security rules concerning the data sharing. This is done by comparing the sensitivity of the resource (e.g. file or storage device) with the clearance of the entity (e.g. user or application). Such rules could include "only members of the accounts department may read or change payroll data", and "classified data may only be accessed by staff with a 'classified' clearance

	level" [MACdefinition]. MAC cannot be overridden without special authorisations or privileges. Contrast with discretionary access control. See Also Discretionary Access Control.
Role Based Access Control (RBAC)	Also referred to as role-based access control. A means of controlling user access by providing applications to users through roles. The role definition contains privileges and security labels needed for the user(s) to carry out permitted operations.
Transition event	A transition event is used in the context of a State flow security model extension (SFSME). Any transition to a new state is represented by a transition event. See Also Event.
Type Enforcement (TE)	Domain type enforcement defines access of domains (subject classes) to types (classes of objects), or other domains by reference to an access matrix. SELinux simplifies this model and also describes domains as types [TEmodel]. See Also Domain and Type Enforcement.
Type	A property of the Domain and Type Enforcement (DTE) security model. A type can be assigned to any object, in the case of the simplified Type Enforcement (TE) model also to any subject. In this work I denote the type as an attribute of both an object and subject. See Also Domain.
Virtual Space (VS)	A separated area of certain system objects (e.g. files, processes, pipes, ..). Used in the ZP Security Framework See Also Type, ZP Security Framework.
ZP Security Framework (ZP)	The ZP Security Framework provides a general purpose access control framework using separated virtual spaces and a definition of their relations. The virtual space abstraction allows implementing various security models (through substitution of the access matrix). Medusa DS9 represents an implementation of the ZP Security Framework. See Also Virtual Space.

Chapter 1. Introduction

1.1. Quo vadis Linux security?

Secure Enhanced Linux (SELinux) was introduced by The National Security Agency at the Linux Kernel Summit in March 2001. SELinux represents a fine-grained, flexible and precise access control mechanism implemented in the Linux kernel (originally using a special patch). Similar strong access control mechanisms have been implemented independently of SELinux (RSBAC and Medusa), various other projects (DTE for Linux) developed their own implementation of a security model. All require a special patch to the kernel.

As a response to the NSA presentation, Linus Torvalds proposed a basic characteristics of a security framework which has been successfully integrated into the Linux kernel branch 2.5.x. He described a set of security hooks that a given security framework should include for access control over kernel objects, kernel security-aware structures and their security attributes. The main goal was to build a security framework suitable for kernel developers' needs. The framework should be an adequate solution for the security modules implementing various security models. Linus also proposed the possibility of the migration of the Linux capabilities to such a module.

1.1.1. LSM Framework

The development of Linux Security Modules (LSM) [LSM] project was initiated by WireX, later a community of the developers has grown. Nowadays the LSM is a part of the latest stable 2.6.x kernel. The LSM kernel implementation provides a general kernel framework to support security modules. In particular, the LSM framework is primarily focused on supporting access control modules, although future development is likely to address other security needs such as auditing. By itself, the framework does not provide any additional security; it merely provides the infrastructure to support security modules. The LSM kernel implementation also moves most of the capabilities logic into an optional security module, with the system defaulting to the traditional superuser logic.

Linus mentioned per-process security hooks in his original remarks as a possible alternative to global security hooks. However, if LSM were to start from the perspective of per-process hooks, then the base framework would have to deal with how to handle operations that involve multiple processes (e.g. kill), since each process might have its own hook for controlling the operation. This would require a general mechanism for composing hooks in the base framework. Additionally, LSM would still need global hooks for operations that have no process context (e.g. network input operations).

LSM also provides a simple mechanism for stacking additional security modules with the primary security module. It defines register_security and unregister_security hooks in the security_operations structure and provides mod_reg_security and mod_unreg_security functions that invoke these hooks after performing some sanity checking. A security module can call these functions in order to stack with other modules. However, the actual details of how this stacking is handled are deferred to the module, which can implement these hooks in any way it wishes (including always returning an error if it does not wish to support stacking). In this manner, LSM again defers the problem of composition to the module.

LSM also adds a general security system call that simply invokes the sys_security hook. This system call and hook permits security modules to implement new system calls for security-aware applications.

The interface is similar to socketcall, but also has an id to help identify the security module whose call is being invoked.

1.1.2. LSM support in security projects

Due to the fact that the LSM has been included to the latest stable kernel, LSM based applications have a better chance to be used as secure implementations.

RSBAC is considered to be the most complete implementation of a RBAC model. The current concept of the LSM unfortunately does not fit the requirements of the RSBAC implementation due to the incompleteness of the LSM and the absence of additional non-access control security mechanisms (e.g. symlink redirection, secure deleting, ignoring of Linux DAC). The LSM represents only additional security access control mechanism. The author of RSBAC Amon Ott specified the list of issues related to the migration of the RSBAC to LSM.

SELinux. In spite of the contradictory statements of the kernel developers, SELinux has been implemented in the Linux kernel since version 2.6.x. It was the beginning of a massive penetration of SELinux in various Linux distributions (Debian, Gentoo, Redhat, Fedora)

Medusa is an implementation of ZP framework [PikulaZelem2001]. At present Medusa does not support LSM natively, but the authors of Medusa consider its use in the future versions for 2.6.x kernel. The penetration of Medusa comparing to SELinux or Grsecurity into security projects is quite low, despite the smart and universal design of Medusa and its ability to implement various security models using virtual spaces.

Grsecurity along with RSBAC does not support LSM in its current version, the reasons are similar - the lack of required security hooks, impossibility to use a security mechanism other than the LSM access control mechanism, etc. For detailed information refer LSM counter-arguments. The current version (2.0) of Grsecurity supports the RBAC model.

DTE for Linux is a simple implementation of the DTE model written in Python and has native LSM support. The project was developed by only one person, thus only a few characteristics of the DTE model have been implemented.

The development version of **LIDS** has native LSM support. LIDS along with Grsecurity do not focus on the implementation of the RBAC or DTE security models, but implement a set of various security protections.

1.2. General-purpose security description of applications

The purpose of LSM was to provide a common security framework for existing security projects. There were no propositions how to unify the security policy representations based on the same security model. There are a lot of security projects based on known security models (DTE, RBAC, MLS). Each of this project uses a different representation of the security policy. There is no generic representation understandable to all of them. The transformation of a security policy from one security project to another is quite a complex problem. Since some security projects use the same security models, we propose a sufficiently general-purpose, robust RBAC and DTE security model description. This description involves object and subject associations and their relations according to a given security model. The description has been proposed considering the existing security policy representations. An XML notation has been used for the description representation which can be transformed to the real security policy configuration. The transformation is realised using security project specific XSL transformation style-sheets.

Chapter 2. Security models

2.1. Existing models

2.1.1. Bell-La Padula (BLP) model

One of the oldest security model proposed by Bell and La Padula for the Multics Operation System. It implements a Mandatory Access Control (MAC).

In the Bell-La Padula model we consider sets, elements of sets, and an interpretation of the elements of the sets [LaPadula1973]

Table 2.1. Elements of the Bell-La Padula Model

Set	Elements	Semantics
S	$\{S_1, S_2, ..., S_n\}$	subjects: processes, programs in execution
O	$\{O_1, O_2, ..., O_m\}$	objects: data, files, programs, subjects, I/O, devices
C	$\{C_1, C_2, ..., C_q\}$ $\{C_1 > C_2 > ... > C_q\}$	classifications: clearance level of subject, classification of an object
K	$\{K_1, K_2, ..., K_r\}$	categories: special access privileges
A	{r,w,e,a,c}	access attributes: read, write, append, execute, and control
RA	{g,r,c,d}	requests elements: g: get, give r: release, rescind c: change, create d: delete
R	$S^+ \times RA \times S^+ \times O \times X$ where $S^+ = S \cup \{ \varphi \}$ and $X = A \cup \{ \varphi \} \cup F$	requests: inputs, commands, requests for access to objects by subjects
D	{ yes, no, error, ?}	decisions
T	{ 1,2, ..., t}	indices: elements of the time set; identification of discrete moments
F	$C^S \times C^0 \times (PK)^S \times (PK)^O$, an arbitrary element of F is written $f=(f_1,f_2,f_3,f_4)$	classification/need-to-know vectors: f1: subject-classification function f2: object-classification function f3: subject-category function f4: object-category function
X	R^T, an arbitrary element of X is written x	request sequences
Y	D^T, an arbitrary element of Y is written y	decision sequences
M	$\{M_1, M_2, ..., M_c\}$ is an $n \times m$ matrix	access matrices: (i,j)-entry of M_k shows S_i's access attributes relative to O_j
V	$P(S \times O \times A) \times M \times F$	states: an arbitrary element of V is written v

Set	Elements	Semantics
Z	V^T	state sequences: an arbitrary element of Z is written z; $z_t$is the t-th state in state sequence z

All users and resources (objects and subjects) have to be classified in levels of confidentiality (subject's clearance and object's classification).

We define a state $v \in V$ as 3-tuple (b, M, f) where:

- $P(S \times O \times A)$ represents a set of all accesses between subjects and objects
- $b \in P(S \times O \times A)$ indicating which subjects have access to which objects in the state v
- $M \in M$ indicating the entries of the access matrix in the state v
- $f \in F$ indicating the clearance level of all subjects, the classification level of all objects, and the needs-to-know associated with all subjects, and objects in the state v

and **compromise state (compromise)** if there is an ordered pair of subjects/objects $(S,O) \in b$ such that

i. $f_1(S) < f_2(O)$ (S's clearance is lower than O's classification) or

ii. $f_3(S) \not\supseteq f_4(O)$ (S does not have some need-to-know category that is assigned to O).

So, we can make the conclusion: $(S,O) \in S \times O$ satisfies the security condition relative to f if:

iii. $f_1(S) \geq f_2(O)$ and

iv. $f_3(S) \supseteq f_4(O)$.

A state $v = (b,M,f) \in V$ is a *secure state* if each $(S,O) \in b$ satisfies the above-mentioned security condition.

Naturally, a state sequence has a compromise if any of its states is a compromise state. According to Bell and La Padula a real secure system doesn't contain a state sequence with compromise.

The Bell-La Padula model treats only confidentiality aspects, not integrity, availability and privacy of data (e.g. a subject with the lowest security level can delete all data in all its categories). It is a good choice to combine this model with other confidentiality-providing models.

2.1.2. Multi level security (MLS) model

The MLS model [MLSdescript] has been developed from the Bell-La Padula (BLP) model [LaPadula1973]. If a subject is multi-level, i.e. its low level differs from its high level, then it is trusted to handle data at any level in its range while maintaining proper separation among the different levels. Multi-level objects may be used for the private state of multi-level subjects and for data sharing between multi-level subjects.

2.1.3. Privacy / Clark Wilson model

The Clark-Wilson model [ClarkWilson1997] has been developed to address security issues in a commercial environment. The model uses two categories of mechanisms to guarantee integrity: well-formed transactions and separation of duty. Well-formed transactions prevent users from manipulating data, thus ensuring the internal consistency of data. Separation of duties prevents authorised users from making improper modifications, thus preserving the external consistency of data by ensuring that data in the system reflects the real-world data it represents.

The Clark-Wilson model differs from subject and object oriented models by introducing a third access element *programs* resulting in what is called an access triple, which prevents unauthorised users from modifying data or programs. In addition, this model uses integrity verification and transformation procedures to maintain internal and external consistency of data. The verification procedures confirm that the data conforms to the integrity specifications at the time the verification is performed. The transformation procedures are designed to take the system from one valid state to the next. The Clark-Wilson model is believed to address all three goals of integrity.

2.1.4. Role based access control (RBAC) model

The Role based access control model has been proposed by D.F.Ferraiolo, D.R.Kuhn, R.Chandramoli. The latest proposed RBAC standard [Proposed-NIST-RBAC] is maintained by National Institute of Standards and Technology

The basic idea of RBAC is to associate the permissions to the roles, not directly to the users. Likewise each user (or subject) is associated with one or more roles. The roles can be mapped according to the roles in the real organisation. Each role contains the permissions that are needed for its correct operation. The change of the subject role follows the change of the user position in the organisation. Similarly, the integration of a new security object with security permissions involves the appropriate modification of the roles.

2.1.4.1. Roles and groups

The idea of separated group of subjects is implemented in many access control systems. The main difference between the role and the group concept is based on the fact that the group contains solely a set of users, while the role can also contain a set of permissions.

The membership of users in UNIX groups is centralised in /etc/group, the permissions of the groups are decentralised. That means an Unix user is able to decide which group he is member of and can read/modify his own files. Whereas both the membership and the permissions of the roles are centralised.

2.1.4.2. The classes of the RBAC model

During the evolution of the RBAC model there were defined 4 classes of the RBAC model:

The classes of the RBAC model

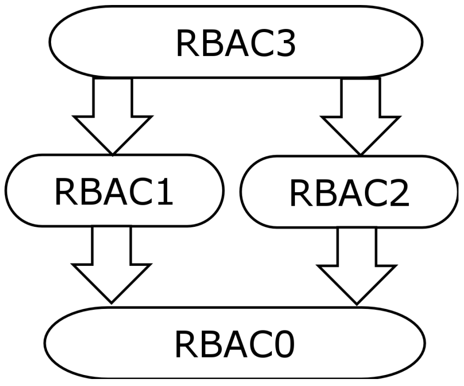
$RBAC_0$
: Core RBAC model. Basic mode with minimal access control system based on roles.

$RBAC_1$
: Hierarchical RBAC model. Involves $RBAC_0$ and role hierarchy. A role can inherit permissions from its parent role.

$RBAC_2$
: Constrained RBAC model. Involves $RBAC_0$ and a set of constraints. Constraints represent some logic conditions applicable to all parts of the RBAC model.

$RBAC_3$
: Unified model. Involves $RBAC_1$ and $RBAC_2$.

Figure 2.1. Relation between the classes of the RBAC model



2.1.4.2.1. Core model $RBAC_0$

Contains three sets of entities:

Subjects
: Entity subject doesn't need to be a person, it can be also a process, service, etc.

Roles
: Users are assigned to roles, permissions are assigned to roles and users acquire permissions by being members of a role. Thus the same user can be assigned to many roles and a single role can have many users. Similarly, for permissions, a single permission can be assigned to many roles and a single role can be assigned to many permissions. Core RBAC includes requirements for user-role review whereby the roles assigned to a specific user can be determined as well as the users assigned to a specific role. Finally, core RBAC requires that users can simultaneously exercise permissions of multiple roles. This precludes products that restrict users to activation of one role at a time.

Permissions

An entity permission represents an operation which is allowed between one or more objects in the system. Entity permissions are all "positive"--meaning, they grant permissions rather than taking them away. Anything that is not explicitly allowed by the permission we assume to be prohibited. Permission can have wide or narrow validity, e.g. permission to read files, use kernel syscall(s), switch security context(s) etc.

In $RBAC_0$ we consider the following relations between entities:

UA (User/subject Assignment)

N to N relation between a set of subjects and a set of roles.

PA (Permission Assignment)

N to N relation between a set of permissions and a set of roles, defines capabilities/rights of each role.

S (Session)

1 to N relation between one subject and many roles. One subset of subject roles is activated after successful user login. User permissions during the session depends on the appropriate activated roles. $RBAC_0$ allows multiple sessions and the possibility of the user to decide which of his roles will be activated or disabled.

2.1.4.2.2. Hierarchical model $RBAC_1$

In addition to $RBAC_0$, $RBAC_1$ provides a relation between each other role in order to ensure the inheritance. The simplest example of role use is e.g. a relationship between an inferior employee and his superior employer. The superior employer is authorised to do all the work of his inferior employee plus a lot of other work the inferior employee can't.

2.1.4.2.3. Constrained model $RBAC_2$

In addition to $RBAC_0$, $RBAC_1$ provides a set of constraints. These constraints are applicable to UA, PA and S relation and also to a set of subjects, roles or permissions. The constraints are predicates involving an information about acceptability of RBAC states.

The most frequent constraint in $RBAC_2$ is the mutual exclusion of the roles. It implies the subject can be member of only one role (from multiple roles), which is exclusive to another role. There are two types of this constraint:

- static (SSD) - there is no possibility to assign two mutual exclusive roles to the subject
- dynamic (DSD) - there is no possibility of the subject to have both mutual exclusive roles during one session

Another important constraint in $RBAC_2$ is the cardinal constraint (capacities of the subjects/roles). A minimum or maximum number of subjects that can be members of a role can be given. Also there can be given a minimum or maximum number of roles a subject can be member of.

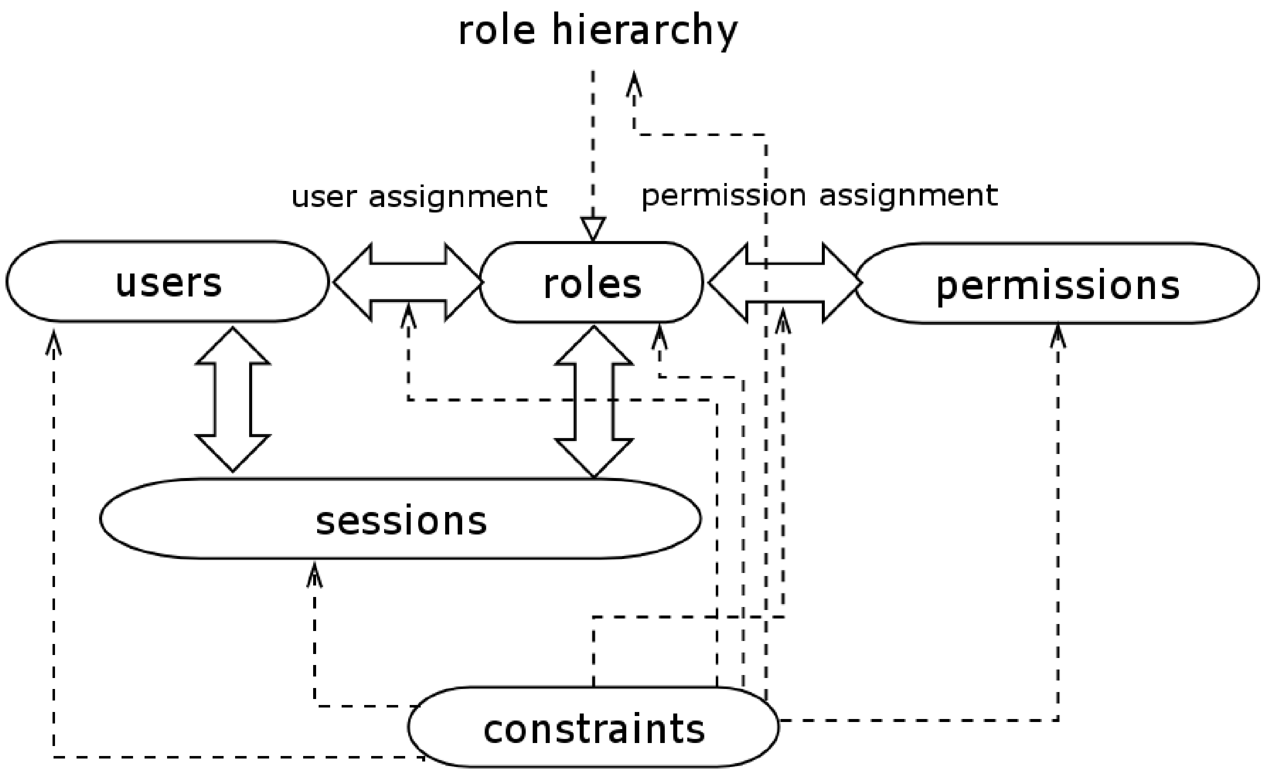
Another constraint of $RBAC_2$ is the constraint allowing the subject to be a member of the role A only if he is also member of the role B.

Similar constraints can be applicable also to an user session, not only UA, PA relation. The constraints of the UA relation in practice require that one human can be just one subject in the RBAC system. Otherwise he can bypass some security constraints.

2.1.4.2.4. Unified model $RBAC_3$

$RBAC_3$ unifies $RBAC_1$ and $RBAC_2$. Thus it involves role hierarchy and a set of constraints.

Figure 2.2. The unified model of $RBAC_3$



2.1.4.3. RBAC control

The RBAC access control model [Ferraiolo92] describes also transactions in addition to subjects and roles. A transaction is expressed as a transformation procedure including necessary data accesses. The most subject activities in a a system are performed through transactions, but not the systems tasks like identification or authentication.

The RBAC model defines 3 rules:

1. **Role assignment -** A subject can execute a transaction only if the subject has selected or been assigned a role

2. **Role authorisation -** A subject's active role must be authorised for the subject

3. **Transaction authorisation -**A subject can execute a transaction only if the transaction is authorised for the subject's active role

Ferraiolo, Cugini and Kuhn [Ferraiolo95] proposed so-called *operation*, which describes the access mode to a set of objects. Then we can consider the roles that are authorised for operation and not for transactions. There is also a difference between users and subjects - a subject is an active entity, who is able to perform user operations at a time and has a set of active roles, for which the user has to be authorised.

2.1.4.4. RBAC administration

Involves administration of:

- User Assignment, Permission Assignment relations
- Role hierarchy
- Set of roles, subjects, permissions and constraints

Using role hierarchy and constraints we are able to decentralise administration of each role and role hierarchy to role administrators. Almighty security administration can set limited privileges to each role administrator.

The detailed information about RBAC administration, administration roles are described in [Proposed-NIST-RBAC].

2.1.5. Domain type enforcement (DTE) model

DTE was originally proposed by Boebert and Kain [Boebert1985].

As with many access control schemes, type enforcement considers a system as a set of active entities - subjects and a set of passive entities - objects. In DTE for UNIX, an access control attribute called *domain* is associated with each subject (Unix process) and another attribute called a *type* is associated with each object (IPC message, file, shared memory, etc.)

In DTE we consider two domain tables:

- **Domain Definition Table (DDT)** represents allowed access modes between domains and types (e.g. read, lock, write, execute)
- **Domain Interaction Table (DIT)** represents allowed access modes between domains (e.g. signal, create)

Analogous to RBAC, all access attempts which are not authorised directly in the tables are denied.

DTE policies can be specified in various languages (DTEL, SELinux TE language etc.)

Commonly a DTE policy should have implemented the following components:

- **Type** - declares one or more object types later used in the DDT
- **Domain** - defines a restricted execution environment composed of three parts:
  1. "entry point" programs, identified by full pathname, that a process has to execute in order to enter the correct domain (e.g. /usr/sbin/sshd, /bin/login)
  2. access rights to types of objects (e.g. read/append to passwd_t)
  3. access rights to subjects in other domains (e.g. transition)
- **Initial_domain** - defines the domain of the first process
- **Assign** - associates a type with one or more files. An assign statement may override another. Assignment can be performed by:
  - implicit typing - DTE can be applied to existing files with no change to file system formats (using regexp to whole directory, ...)
  - explicit typing - security types are stored one-to-one with files on disk

2.1.6. ZP Security Framework model

ZP was proposed by Marek Zelem and Milan Pikula [PikulaZelem].

This youngest VS model takes advantage of many existing models using so-called *virtual spaces*. The authors of the ZP Security Framework described various transformation methods providing the substitution of existing security models and its properties to virtual spaces (VS model).

2.1.6.1. VS concept

The state of the system is defined as a triple (S, O, M) where S is a set of subjects, O is a set of objects and M is the access matrix with "subject" rows and "object" columns. Each element of M[s,o] represents the access rights of subject *s* for object *o*. Access rights are a subset of a finite set of access rights A (a ϵ A) (e.g. Medusa as an implementation of ZP Security framework model provides READ, WRITE, SEE, CREATE, ERASE, ENTER and CONTROL rights). Naturally the presence of particular particular a ϵ A in the cell of table M[s,o] defines the right for subject *s* to perform access of type a on object *o*.

The VS model provides separation of the objects and subjects into a finite number of domains called virtual spaces (VS).

Each object o ϵ O in the system can be a member of zero, one or more virtual spaces. This is defined by the function OVS: $O \rightarrow VS^*$.

Each subject s ϵ S is assigned a set of abilities, one for each access type a ϵ A. Ability can be set of zero, one or more virtual spaces. This is defined by the function SVS_a: $S \rightarrow VS^*$.

Access of type $a \in A$ of the subject $s \in S$ on the object $o \in O$ is granted when the object and the subject share at least one common virtual space for the given access type: $SVS_a(s) \cap OVS(o) \neq \emptyset$

The state of the system is changed by changing the abilities of subjects or membership of objects.

2.1.6.2. VS properties

Any state of the system with subjects, objects and an access matrix can be translated to the VS model in case we have a sufficient number of virtual spaces. The result of the OVS and SVS_a functions is a set of virtual spaces. The presence or absence of each virtual space $vs \in VS$ in the set can be represented as an array of boolean values, making it possible to describe some operations using boolean algebra.

2.2. Model Implementations

2.2.1. RSBAC

RSBAC [RSBACImpl] has been written and is still maintaining by Amon Ott.

RSBAC is based on the *Generalised Framework for Access Control* (GFAC) [LaPadula1990], which describes a general framework approach to separate access control between enforcement, decision, access control and authorities who are allowed to modify that data. The design followed the suggestions in [LaPadula1995] - how to implement GFAC in *nix system and implemented some access control models (MAC, FC and SIM) described there.

RSBAC involves the first Linux implementation (since December 1998) of the Role Compatibility (RC) model. The RSBAC framework provides a generic infrastructure for security model implementations including persistent list management. It groups access objects into so-called target types, e.g. FILE, DIR, IPC. Similarly, network access is controlled through network templates providing dynamic network objects.

RSBAC provides independent model support and runtime module registration (REG).

RSBAC includes the following models [AmonOtt2002]:

- *Mandatory Access Control (MAC)* - based on the Bell-La Padula model
- *Functional Control (FC)* - role based model of functional control assigns one role to each user, FC can be easily expressed with RC model, so this model is obsolete
- *Security Information Modification (SIM)* - this model protects data of type "security information". Only users with role "security office" get write access to those objects. SIM can be also easily expressed using the RC model, so it's kind of obsolete.
- *Simone Fischer-Hubner's Privacy Model (PM)* - developed by Simone Fischer-Hubner, this model follows the rules of the Federal German Privacy Law and the EU directive on privacy. This model should be used for storage and processing of personal data.
- *Malware Scan (MS)* - this is not really an access control model, but rather a system protection against execution, reading and transmission of malware infected files.

- *File Flags (FF)* - this model defines some access flags for files, FIFOs, symlinks and directories. These flags are checked on every access to the given target types. Only users in the 'security_officer' system_role can change the flags.

- *Role Compatibility (RC)* - is a role based model, it means that every user has a default role, which is inherited by all his/her processes. Based on the current role, access to objects of certain types is granted or denied. The role can be changed by changing the process owner, by the process via system call (only "compatible" roles allowed) or by executing a specially marked executable (using initial_role or force_role, need not be "compatible"). Creation of new objects is a special case in every access control model. Here, every role has entries for the types of new objects, as well as entries for type setting behaviour on execution and change of process owner. Additionally, all these entries have a "no_create" special value, which disallows such requests.

- *Authentication Module (AUTH)* - this module can be seen as a support module for all others. It restricts CHANGE_OWNER on process targets (setuid) for a process: the request is only granted if the process has either the auth_may_setuid flag set or the target user ID is in its capability set. The auth_may_setuid flag and the capability set are inherited on execute from the program file.

- *Access Control Lists Module (ACL)* - access Control Lists specify which subject (user, RC role or ACL group) may access which object (of an object type) with which requests (usual RSBAC requests and ACL specials).

- *Linux Capabilities (CAP)* - this model allows to define a minimum and maximum capability set for both users and programs. Program settings override user settings, and minimum settings override maximum settings. This module can be used to restrict rights of programs run by root, add root rights to normal users or programs run by them.

- *JAIL* - this module provides a new call rsbac_jail, which makes a chroot call (with chdir("/")) and adds further restrictions on the calling process and all subprocesses.

2.2.2. SELinux

SELinux[SELinuxImpl] represents one of the most complex security framework implementations. SELinux uses the so-called Flask architecture, which provides flexible support for mandatory access control policies. In a system with mandatory access controls, a security label is assigned to each subject and object. Every access from a subject to an object or between two subjects has to be authorised by the policy based on these labels. The Flask architecture separates the definition of the policy logic from the enforcement mechanism. SELinux *security server*, which provides an interface for obtaining security policy decisions, is implemented as a kernel subsystem.

In the Flask architecture components in the operating system that enforce the security policy are referred to as *object managers*. Object managers are modified to obtain security policy decisions from the security server and to apply these decisions to label and control access to their objects. Process management, file-system, socket IPC, etc. are object managers in the SELinux implementation.

2.2.2.1. Flask Architecture [LoscoccoNSATR2001]

- provides an access vector cache (AVC) component that stores the access decision computations provided by the security server for subsequent use by the object managers. The AVC component also supports the revocation of permissions. The Flask AVC provides an interface to the security server for managing the cache as needed for policy changes. When the AVC receives a policy change notification, it updates its own state and then invokes callback functions registered by the object managers to update any permissions retained in the state of the object managers. After updating the state of the object managers and the state of the AVC to conform the policy change, the AVC notifies the security server that the transition to the new policy has been completed.[1]
- defines two policy-independent data types for the security label - a *security context* as a representation of the security label and a *security identifier (SID)* as an integer that is mapped by the security server to a security context. Security labels are binded to the system objects by object managers (SIDs are binded to kernel objects).
- specifies the interfaces provided by the security server to the object manager. The implementation of the security server, including any policy language it may support, are not specified by the architecture, [2]

When a Flask object manager requires a label for a new object, it consults the security server to obtain a labelling decision based on the label of the creating subject, the label of a related object, and the class of the new object. For program execution, the Flask process manager obtains the label for the transformed process based on the current label of the process and the label of the program executable. For file creation, the Flask file system manager obtains the label for the new file based on the label of the creating process, the label of the parent directory, and the kind of file being created. [3]. For objects where there is only one relevant SID, object managers do not consult the security server (pipes, file descriptions and sockets inherit the SID of the creating process, and output messages inherit the SID of the sending socket).

Figure 2.3. Flask interface and example call to obtain a security label

```
int security_transition_sid (security_id_t ssid, security_id_t tsid,
security_class_t tclass, security_id_t *out_sid);
```

Object managers consult the AVC to check permissions based on a pair of labels and object class, and the AVC obtains access decisions from the security server as needed. Each object class has a set of associated permissions represented by a bitmap called an *access vector*. Flask defines a distinct permission

[1]In SELinux, permissions are revalidated on use, such as permissions for reading and writing files and permissions for communicating on an established connection. Consequently, policy changes for these permissions are automatically recognised and enforced without the need for object manager callbacks. The *security_load_policy* call may be used to read a new policy configuration from a file.

[2]SELinux security server defines a security policy that is a combination of Type Enforcement (TE), role-based access control RBAC and optionally multi-level security (MLS). SELinux security server also defines a security context as containing an user identity, a role, a type and optionally a MLS level or range. The security server only provides SIDs for security contexts with legal combinations of user, role, type and level or range.

[3]The SELinux security server may be configured to automatically change the role or domain attributes of a process based on the role and the original process and the type of the program or use specified types for new files based on the domain of the process, the type of the parent directory, and the kind of file.

for each service, and when a service accesses multiple objects, Flask defines a separate permission to control access to each object.[4]

2.2.2.2. Security Mechanisms

This sections describes the security mechanisms defined by the Flask architecture and the SELinux implementation of these mechanisms (process, file and socket controls).

- **Process Controls**

Table 2.2. SELinux process object class permissions

Permission	Description
execute	Execute
transition	Change label
entrypoint	Enter via program
sigkill	Signal KILL
sigstop	Signal STOP
sigchld	Signal CHILD
signal	Signal
fork	Fork
ptrace	Trace
getsched	Get Schedule info
setsched	Set Schedule info
getsession	Get session
getpgid	Get process group
setpgid	Set process group
getcap	Get capabilities
setcap	Set capabilities

- The process *execute* permission is used to control the ability of a process to execute from a given executable image. The permission is checked between the label of the transformed process and the label of the executable on every program execution. The *process execute* permission is distinct from the *file execute* permission which is used to control the ability of a process to initiate the execution of a program.
- The *transition* permission is used to control the ability of a process to transition from one SID to another.
- The *entrypoint* permission is used to control which programs may be used as the entry point for a given process SID. This permission is only checked when a process transitions to a new SID.

[4]For example, when a file is unlinked, Flask checks *remove_name* permission to the directory and *unlink* permission to the file.

- The *sigkill, sigstop, sigchld* permissions are used to control the SIGKILL, SIGSTOP and SIGCHLD signals, the permission *signal* is used to control the remaining signals
- The *ptrace* permission is used to control the ability of a process to trace another process
- The *getsched, setsched, getsession, getpgid, setpgid, getcap, setcap* are used to control the ability of a process to observe or modify the corresponding attributes of another process.

In addition to these permissions, SELinux provides an equivalent permission for each Linux capability allowing the security policy to control the use of capabilities based on the SID of the process.

- **File Controls**

Table 2.3. SELinux permissions for the open file description object class

Permission	Description
create	Create
getattr	Get attributes
setattr	Set attributes
inherit	Inherit across execve
receive	Receive via IPC

Since open file descriptions may be inherited across *execve* or transferred through UNIX socket IPC, SELinux labels and controls open file descriptions. An open file description is labelled with the SID of its creating process. We distinguish the label of an open file description and the label of the file it references [5].

Table 2.4. SELinux file-system object class permissions

Permission	Description
mount	Mount
remount	Change options
getattr	Get attributes
relabelfrom	Relabel from
relabelto	Relabel to
transition	Transition
associate	Associate file

SELinux binds security labels to files and directories and controls access to them. It stores a persistent labelling table in each file system that specifies the security label for each file and directory in that file system. For efficient storage, SELinux assigns an integer value referred to as a *persistent SID*

[5]A read operation on a file changes the file offset in the open file description, so it may be necessary to prevent a process from reading a file using an open file description received or inherited from another process even though the process is allowed to directly open and read the file.

(PSID) to each security label associated with an object in a file system. The persistent labelling table is partitioned into a mapping between each PSID and its security label and a mapping between each object and its PSID[6].

Table 2.5. SELinux pipe and file object classes permissions

Permission	Description
read	Read
write	Write or append
append	Append
poll	Poll/select
ioctl	IO control
create	Create
execute	Execute
access	Check accessibility
getattr	Get attributes
setattr	Set attributes
unlink	Remove a file, directory or link
link	Create hard link
rename	Rename a file, directory or link
lock	Lock or unlock
relabelfrom	Relabel from
relabel to	Relabel to
transition	Transition

SELinux also introduces additional permissions for mounting and directory operations.

[6]The mapping between each PSID and its security label is implemented using regular files in a fixed subdirectory of the root directory of each file system. The mapping between each object and its PSID is implemented by storing PSID in an unused field of the on-disk inode.

Table 2.6. SELinux additional permissions for the directory object class

Permission	Description
add_name	Add a name
remove_name	Remove a name
reparent	Change parent directory
search	Search
rmdir	Remove
mounton	Use as mount point
mountassociate	Mount and associate

The process has to have *mounton* permission to the mount point directory and *mount* permission to the file system. It also requires that the *mountassociate* permission be granted between the root directory of the file system and the mount point directory.

- **Socket Controls** in SELinux represent control over socket IP through a set of layered controls [TCPIPlayers], messages, nodes and network interfaces. At the socket layer[7], SELinux controls the ability of processes to perform operations on sockets. At the transport layer[8], SELinux controls the ability of sockets to communicate with other sockets. At the network layer[9], SELinux controls the ability to send and receive messages on network interfaces and the ability to send messages to nodes and to receive messages from nodes. SELinux also controls the ability of processes to configure network interfaces and to manipulate the kernel routing table.

[7]Sockets and Application Layer - http://www.intercode.com.au/jmorris/lsm-ols2002-html/node17.html

[8]Transport Layer (IPv4) - http://www.intercode.com.au/jmorris/lsm-ols2002-html/node19.html

[9]Network Layer (IPv4) - http://www.intercode.com.au/jmorris/lsm-ols2002-html/node20.html

Table 2.7. SELinux socket permissions

Socket permissions			
Permission		**Description**	
bind		Bind name	
connect		Initiate connection	
listen		Listen for connections	
accept		Accept a connection	
getopt		Get socket options	
setopt		Set socket options	
shutdown		Shut down connection	
recvfrom		Receive from socket	
sendto		Send to socket	
recv_msg		Receive message	
send_msg		Send message	
name_bind		Use port or file	
Additional TCP socket permissions	**Additional RAW/UDP socket permissions**	**Additional Unix stream socket permissions**	**Description**
connectto		connectto	Connect to server socket
newconn		newconn	Create new socket for connection
acceptfrom		acceptfrom	Accept connection from client socket
node_bind	node_bind		Bind to node

Table 2.8. SELinux node and interface permissions

Node/interface permissions	
Permission	**Description**
tcp_recv	Receive TCP packet
tcp_send	Send TCP packet
udp_recv	Receive UDP packet
udp_send	Send UDP packet
rawip_recv	Receive Raw IP packet
rawip_send	Send Raw IP packet
Additional node permissions	**Description**
enforce_dest	Enforce destination socket

The *netlink, packet and key sockets* have the same permissions as the simple socket. The *tcp sockets* have additionally *connectto, newconn, acceptfrom and node_bind* permissions. The *Unix stream sockets* have also *connectto, newconn and acceptfrom* permissions. Sockets effectively serve as communications proxies for processes in the SELinux control model and are by default labelled with the label of the creating process. A process may create and use a socket with a different label to perform socket IPC with a different source security label. SELinux allows the security policy to distinguish between clients and server for stream socket connections through the *connectto* and *acceptfrom* permissions, also allows to base decisions on the kind of the socket through the use of object classes or on the message protocol through the per-protocol node and network interface permissions.

2.2.3. Domain and Type Enforcement for Linux

This simple implementation [DTE-for-Linux] of Domain and Type Enforcement follows the description in the papers [DTE-Unix-Prototype] and [Confining-root-program-with-DTE].

The DTE policy server is implemented as a kernel module called *DTE plug*, loading and manipulating the DTE policy is implemented using python scripts `loaddte.py` and `apply.py`. Only a few aspects of the DTE model are implemented (e.g. basic DTE matrix). The appropriate user domains on login are set through the `pam_dte.so` PAM module. The DTE policy is very similar to the DTEL language. The author of DTE for Linux tried to write this project to be cross-platform (there is also a Windows version available). Currently this project seems to be stalled.

2.2.4. Grsecurity

Grsecurity is a suite of patches that is an attempt to improve Linux security. According to [GrsecurityStudy], the creator of Grsecurity project, this suite meets four goals:

- Configuration free operation
- Protections against all kinds of address space modification bugs
- Rich access control list system and many auditing systems
- Operation on multiple processor architectures and operating systems

Since version 2.0, Grsecurity supports the *Role-Based Access Control* model. There is a possibility to define user, group and special roles, role transition tables, IP-based roles, non-root access to special roles and special roles that require no authentication.

The `gradm2` tool allows *full learning mode* which allows to fully automatically generate RBAC users and roles. An administrator doesn't need to manually edit the RBAC policy.

2.2.5. Medusa DS9

Medusa DS9 security system [MedusaImpl] represents a ZP security framework [ZelemPikula2001] implementation.

2.2.5.1. Medusa architecture

Medusa consists of two basic parts - a small patch to the Linux kernel and a user space daemon called "Constable". Constable is the user space implementation of an authorisation server, which simplifies the functionality of the kernel part of Medusa. Communication between Constable and kernel is realised through the fast special device "/dev/medusa". When the kernel needs confirmation to execute some operation, it writes a request to this device, suspends the execution of the current process and wakes up Constable which reads the request, chooses a response (depending on its configuration) and sends it back to the kernel. The kernel gets the data, wakes up the process and based on received result determines the result of the operation. Constable is also able to send certain commands to the kernel (even if the kernel doesn't require them), which are then executed by the kernel. The Constable and the kernel use the specific communication protocol, so the only thing that is needed in order to implement a full-featured authorisation server is the knowledge of this protocol. Constable is an example of such an authorisation server.

2.2.5.2. Medusa kernel framework

The current implementation of Medusa in the kernel allows

- full access control to any file in the system (via VFS)
- ability to redirect access to the selected file or another one
- complete control of signal sending/receiving
- direct control of important process actions
- control of execution of any syscalls for specified processes
- every process or file is a member of a specified virtual subsystems and every process has assigned access rights to VS. (It is possible to completely hide processes or files from other processes this way.)
- every process has a login uid (luid), which is set only on the first call of set{re,s}uid
- ability to force execution of specified code for any process
- low level control of any system call

2.2.5.3. Constable authorisation server

Constable is an authorisation server implementation of the Medusa DS9. The security policy configuration of the Constable is written in a special language very similar to standard C. This language is robust enough to express the above-mentioned low-level controls. Due to the complexity of Constable, we decided to describe only the aspects closely connected to proposed transformations:

Declaration and definition of virtual spaces (VS)

Virtual spaces (VS) are named subsets of k-objects [10]. VS is defined as a pathname in tree. Constable makes possible to define VS not only as a pathname, but also as a complete subtree with some subdirectories possibly excluded.

Figure 2.4. Definition of virtual space (with implicit declaration)

```
[primary] space "name" = <p> [ (+|-) <p> ...]

name - name of virtual space
p - can be a pathname to a given object, recursive pathname (a pathname
that  involves full subtree) or another declared space
sign + or - defines whether a given pathname should be appended
to the VS or excluded
```

Each node (object) does not need to be a member of any virtual space, but it can also be a member of many virtual spaces. A special case is the so-called primary virtual space, where a node can belong only to exactly one primary virtual space. In practice, the primary virtual space is used for the active domain of process.

Access types between VS

We distinguish 7 access types between virtual spaces:

- **READ** - reading or receiving information from the object of operation
- **WRITE** - writing or sending information to the object of operation
- **SEE** - permission to verify the existence of the object of operation
- **CREATE** - permission to create the object of operation
- **ERASE** - permission to erase the object of operation
- **ENTER** - permission to transit to VS of the object of operation
- **CONTROL** - permission to control the object of operation

The Constable implements all these access types. If permitted, the Constable can use the ENTER permission for mapping k-objects to trees. The Linux kernel uses only three access types (READ, WRITE and SEE). Remaining access types are used by additional modules (such as a RBAC module, which uses all of above-mentioned access types for RBAC administration control).

[10]k-object means any kind of object defined by kernel associated with a specific k-object tree (file, process, etc.), where is determined by its type. k-object trees define Constable name-space.

Figure 2.5. Access types mapping to VS

```
<space1> <type> <space2> [, [<type>] <space2> ... ]

space1 - a virtual space which receives permission
type - a type of obtained permission
space2 - a virtual space which can be accessed from space1 using the
received permission
```

Handling functions

The primary purpose of the handling functions in Constable is to provide a desired modification of a given object and/or change its association with VS according to a system event. The secondary purpose is to provide decisions whether to allow or deny certain operations.

Figure 2.6. Handling function definition

```
<space1> <op> [:<decision_list>] [<space2>] { <cmd> .. }

space1 - a virtual space of the subject of operation
space2 - a virtual space of the object of operation (not used when
an operation does not have an object)
op - an operation
decision_list - can have one of following values:
VS_ALLOW - the operation has been allowed by the VS model in the kernel
(the handling function can decide whether to allow or deny the given
operation)
VS_DENY  - the operation has been denied by the VS model in the kernel
(the handling function can not allow the given operation)
NOTIFY_ALLOW - the operation has already been allowed by the relevant
VS_ALLOW function, the handling function is only notified about
the result of the operation
NOTIFY_DENY - the operation has already been denied by the relevant
VS_DENY or VS_ALLOW function, the handling function is only notified
about the result of the operation

cmd - a body (program) of the handling function, the program can return
one of these values:

ALLOW - allow the operation completely ignoring Unix DAC
DENY  - deny the operation
SKIP  - do not execute the operation but report a success
OK    - let Unix DAC to determine the result of the operation
```

Example 2.1. Use of the `fexec` operation in handling function

```
* fexec:NOTIFY_ALLOW ping_exec_t {

process.ecap = process.ecap + CAP_SETUID + CAP_NET_RAW;
process.pcap = process.pcap + CAP_SETUID + CAP_NET_RAW;
process.icap = process.icap + CAP_SETUID + CAP_NET_RAW;

enter_domain ("ping_t");
log_proc ("ping_t");
}
```

Chapter 3. Existing grammars of the security models

3.1. Domain Type Enforcement Language DTEL

One of the first language used for describing DTE [DTE-Unix-Prototype1995] that has been applied on UNIX* systems.

DTEL uses only a few keywords, supports three modes of the typing (implicit at, implicit under and explicit typing), domain and type definition, selection of the initial domain.

Example 3.1. DTEL Policy

```
type unix_t,      /* normal UNIX files, programs, etc. */
specs_t,     /* engineering specifications */
budget_t,    /* budget projections */
rates_t;     /* labor rates */
#define DEFAULT   (/bin/sh), (/bin/csh), (rxd->unix_t) /* macro */

domain engineer_d     = DEFAULT, (rwd->specs_t);
domain project_d      = DEFAULT, (rwd->budget_t), (rd->rates_t);
domain accounting_d   = DEFAULT, (rd->budget_t), (rwd->rates_t);
domain system_d       = (/etc/init), (rwxd->unix_t), (auto->login_d);
domain login_d        = (/bin/login), (rwxd->unix_t), (exec->engineer_d, \
project_d, accounting_d);

initial_domain       system_d;       /* system starts in this domain */

assign -r            unix_t          /* default for all files */
assign -r            specs_t         /projects/specs;
assign -r            budgt_t         /projects/budget;
assign -r            rates_t         /projects/rates;
```

3.2. DTE for Linux

DTE for Linux uses a very similar grammar to DTEL [DTE-Unix-Prototype1995]. It also allows the definition of a default domain, grouping of various domains, sophisticated access permissions to each type (using the keyword *access*), the ability to send/receive signals (using the keyword *signal*).

The author of the paper [DTE-for-Linux2000] also proposed a working implementation of DTE. Unfortunately this project has been stalled and due the broken web links, there was no available documentation and working modules during the writing of this thesis.

Example 3.2. DTE for Linux - type and domain definition (login/getty)

```
Module login

# domains getty_d login_d
# types login_et getty_et
# groups login_domains_grp

type login_et
access all rx
epath /bin/login
end

type getty_et
access all rx
epath /sbin/mingetty
end

domain getty_d
entries getty_et

domain in boot_d auto
domain out login_d auto

signal in login_d 14,17
signal in boot_d 0
signal out boot_d 14,17
signal out login_d 0
end

domain login_d
entries login_et

domain in getty_d auto
domain out user_domains_grp exec

signal out getty_d 14,17
signal in user_domains_grp 14,17
signal out user_domains_grp 0
signal in getty_d 0
end

group domain login_domains_grp extend
import login_d getty_d
end

End
```

Example 3.3. DTE for Linux - DTE definition (login/getty)

```
spec_domain login_d \
(1 login_et)\
(23 rld->init_t rld->root_dir_t rwxlcd->xdm_out_et rwlcd->dev_t rx->su_et
r->shadow_t r->passw_t rxld->bin_t rx->passw_et rx->reboot_et rx->getty_et
rlxd->lib_t rwlcd->log_t rwxlcd->sysfs_t rxld->conf_t rxld->sbin_t
rld->user_t rld->base_t rx->login_et rx->shell_t rld->root_t rwxlcd->tmp_t
rwxlcd->varrun_t) \
(3 exec->root_d exec->user_d auto->reboot_d)\
(3 0->root_d 14,17->getty_d 0->user_d)
spec_domain getty_d \
(1 getty_et)\
(23 rx->reboot_et rwlcd->dev_t rwxlcd->xdm_out_et rld->root_dir_t
r->shadow_t rlxd->lib_t \
r->passw_t rxld->bin_t rld->base_t rld->user_t rx->passw_et rxld->sbin_t \
rwxlcd->varrun_t rwxlcd->sysfs_t rld->init_t rx->getty_et rx->su_et
rwlcd->log_t rxld->conf_t rld->root_t rx->shell_t rwxlcd->tmp_t
rx->login_et) \
(2 auto->login_d auto->reboot_d)\
(2 0->login_d 14,17->boot_d)
assign -e login_et /bin/login
assign -e getty_et /sbin/mingetty
```

3.3. XML RBAC

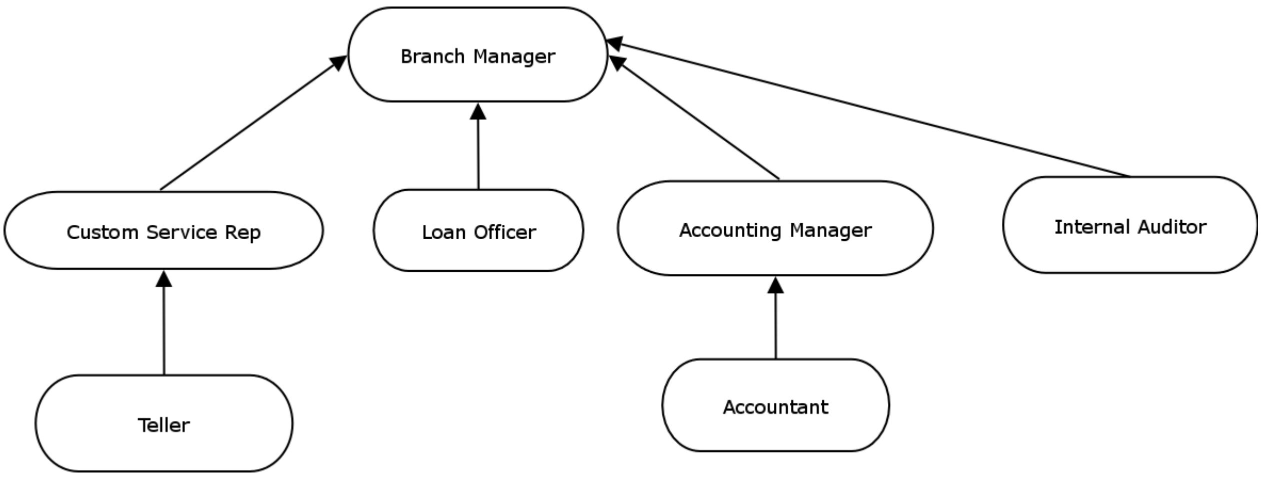
Ramaswamy Chandramouli from NIST described [Chandramouli] the Enterprise-Wide RBAC policy using XML and proposed the appropriate document type definition (DTD).

The DTD has been proposed with regard to:

- *Expressiveness* - requirement to cover the various RBAC model constructs
- *Flexibility* - it would be preferable if the DTD is generic enough to be used for describing most common RBAC models
- *Document-Readability* - the XML document should be readable and hence the logic of the RBAC implementation program that parses this document is not unduly complicated.

The following examples demonstrate a Role hierarchy of Bank Database Application (BANKDB), its proposed Document Type Definition (DTD) and a sample of a possible configuration (XML).

Example 3.4. Role graph for Bank Database Application (BANKDB)



Example 3.5. Bank Database Application (BANKDB) - DTD

```
<!ELEMENT Role_Graph (Application , (role )* )*>
<!ELEMENT Application (DB_Name , Server )>
<!ELEMENT DB_Name (#PCDATA )>
<!ELEMENT Server (#PCDATA )>
<!ELEMENT role (Name , Cardinality? , (Parent_Role?)* , (Child_Role?)* ,
(SSD_Role?)* , (DSD_Role?)* )> <!ELEMENT Name (#PCDATA )>
<!ELEMENT Cardinality (#PCDATA )>
<!ELEMENT Parent_Role (#PCDATA )>
<!ELEMENT Child_Role (#PCDATA )>
<!ELEMENT SSD_Role (#PCDATA )>
<!ELEMENT DSD_Role (#PCDATA )>
```

Example 3.6. Bank Database Application (BANKDB) - XML

```
<?xml version="1.0" ?>
<!DOCTYPE Role_Graph SYSTEM "RBAC.dtd">
<Role_Graph>
<Application>
<DB_Name>Bank Corporate Database</DB_Name>
<Server>Solaris</Server>
</Application>
<role>
<Name>Branch_Manager</Name>
<Cardinality>1</Cardinality>
<Child_Role>Customer_Service_Rep </Child_Role>
<Child_Role>Loan_Officer </Child_Role>
<Child_Role>Accounting_Manager </Child_Role>
<Child_Role>Internal_Auditor </Child_Role>
</role>
............................
</Role_Graph>
```

3.4. SELinux policy

SELinux uses one of the most robust grammars able to describe various aspects of the TE and RBAC models. The grammar was originally documented informally by Peter Loscocco and Stephen Smalley [LoscoccoNSATR2001]. The example policy configuration was originally documented by Stephen Smalley and Timothy Fraser [SmalleyNAITR2001].

The SELinux policy grammar has been proposed using *Flex* (fast lexical generator) and the policy parser has been generated using *Bison* (GNU parser generator). *GNU m4* macro processor has been used for user-defined macros.

The policy language grammar is specified in the `module/checkpolicy/policy_parse.y` file. M4 macro definitions can be found in the `policy/macros/global_macros.te` file.

A policy configuration involves the following components:

- Flask definitions
- RBAC and TE declarations and rules
- User and role declarations
- Constraint definitions
- Security context specifications

3.4.1. Flask definitions

These definitions involve a declaration of the security classes, initial SIDs and access vectors (permissions for each class). This information is not specific to any particular security model and is only related to the Flask architecture.

3.4.2. RBAC and TE declarations

These rules declare the roles, domains, types and define the labelling and access vector rules for the TE and RBAC models. The policy language allows TE and RBAC configuration to be mixed.

Example 3.7. Syntax of the TE and RBAC declarations and rules

```
te_rbac -> te_rbac_statement | te_rbac te_rbac_statement
te_rbac_statement -> te_statement | rbac_statement
te_statement -> attrib_decl |
type_decl |
type_transition_rule |
type_change_rule |
te_av_rule |
te_assertion
rbac_statement -> role_decl |
role_dominance |
role_allow_rule
```

TE statements

Attribute declaration

A type attribute is used to identify a set of types with a similar property. Each type can have any number of attributes and each attribute can be associated with any number of types. The attribute association is expressed in the type declaration. Before the first use of an attribute in a type declaration, it has to be explicitly declared.

Example 3.8. SELinux - attribute declaration

```
attribute domain;
attribute privuser;
attribute privrole;
```

Type declaration

Every type in the TE model has to be declared. A type declaration specifies its name for the type, an optional set of aliases and an optional set of attributes. Aliases represent only shorthand forms/synonyms for types, the security server always uses the primary name.

Example 3.9. SELinux - type declaration

```
type sshd_t, domain, privuser, privrole, privlog, privowner;
type sshd_exec_t, file_type, exec_type, sysadmfile;
type sshd_tmp_t, file_type, sysadmfile, tmpfile;
type sshd_var_run_t, file_type, sysadmfile, pidfile;
```

There is no difference in the SELinux policy language between types and domains, the DTE-specific domain is a SELinux type with a *domain* attribute, the DTE-specific type is a SELinux type with other attributes.

TE transition rules

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

Example 3.10. SELinux - TE transition rules

```
type_transition initrc_t sshd_exec_t:process sshd_t;
type_transition sshd_t tmp_t:{ dir file lnk_file
sock_file fifo_file } sshd_tmp_t;
type_transition sshd_t shell_exec_t:process user_t;
```

The SELinux policy language implements *class macros* which group several classes with similar meaning (e.g. dir_file_class_set, file_class_set, notdevfile_class_set, devfile_class_set, socket_class_set, dgram_socket_class_set, stream_socket_class_set, unpriv_socket_class_set) and defines some useful macros **domain_auto_trans** for domain transitions and **file_type_auto_trans** for file type transitions.

Example 3.11. SELinux - TE transition rules (using macros)

```
domain_auto_trans(initrc_t, sshd_exec_t, sshd_t)
file_type_auto_trans(sshd_t, tmp_t, sshd_tmp_t)
domain_auto_trans(sshd_t, shell_exec_t, user_t)
```

TE change rules

The SELinux policy language, according to the TE specification, supports TE change rules. These rules are not used by the kernel, but can be obtained and used by security-aware applications through the *security_change_sid* system call. A TE change rule specifies the new type to use for a relabelling operation based on the domain of a user process, the current type of the object, and the class of the object.

Example 3.12. SELinux - TE change rules

```
type_change user_t tty_device_t:chr_file user_tty_device_t;
type_change sysadm_t tty_device_t:chr_file sysadm_tty_device_t;
type_change user_t sshd_devpts_t:chr_file user_devpts_t;
type_change sysadm_t sshd_devpts_t:chr_file sysadm_devpts_t;
```

TE access vector rules

One of the base elements of the TE model is the TE access matrix. The SELinux policy describes the TE access matrix using TE access vector rules. Rules can be specified for each kind of access vector, including the *allowed*, *auditallow* (allow auditing of the rule) and *auditdeny* (do not allow auditing of the rule).

Example 3.13. SELinux - TE access vector rules

```
allow sshd_t sshd_exec_t:file { read execute entrypoint };
allow sshd_t sshd_tmp_t:file { create read write getattr setattr link
unlink rename };
allow sshd_t user_t:process transition;
```

The first parameter represents the source type, the second represents the target type with the appropriate class and the third the set of allowed permissions.

Similar to TE transition rules, some *class macros* can be used in the class field of a TE access vector rule.

TE access vector assertions

The SELinux TE policy language allows the policy writer to define a set of access vector assertions which are checked by the policy compiler. An access vector assertion specifies permissions which should not be used in an access vector for a given type pair and class. Assertions are used to detect errors in the TE access vector rules that may be not be evident from a manual inspection of the rules.

Example 3.14. SELinux - TE access vector assertions

```
neverallow domain ~domain:process transition;
neverallow ~{ kmod_t insmod_t rmmod_t ifconfig_t } self:capability
sys_module;
neverallow local_login_t ~login_exec_t:file entrypoint;
```

RBAC statements

Role declarations and dominance

A role declaration specifies the name of the role and the set of domains for which the role is authorised.

The SELinux policy language uses the *types* keyword in the role declaration because the SELinux TE model uses a single type abstraction. Multiple role declarations can be specified for a single role, in which case the union of the type will be authorised for the role.

Example 3.15. SELinux - RBAC role declaration

```
role system_r types { kernel_t initrc_t getty_t klogd_t };
role user_r types { user_t user_netscape_t };
role sysadm_r types { sysadm_t run_init_t };
```

Role dominance definitions can optionally be used to specify a hierarchy among roles. A role automatically inherits any domains that are authorised for any role that it dominates in the hierarchy.

Role allow rules

Role allow rules specifies authorised transitions between roles based on a pair of roles. Unlike domain transitions, the RBAC policy does not control role transitions based on the type of the entrypoint program.

Example 3.16. SELinux - RBAC role allow rules

```
allow system_r { user_r sysadm_r };
allow user_r sysadm_r;
allow sysadm_r system_r;
```

3.4.3. User Declarations

The user declarations define each user recognised by the policy and specifies the set of authorised roles for each of these users.

The user identity attribute in the security context remains unchanged by default when a program is executed. Security-aware applications, such as the modified `login` or `sshd` programs, can explicitly specify a different user identity using the *execve_secure* system call.

Example 3.17. SELinux - User declarations

```
user system_u roles system_r;
user root roles { user_r sysadm_r };
user jdoe roles user_r;
```

3.4.4. Constraint Definitions

The constraint definitions specify additional constraints on permissions in the form of boolean expressions that must be satisfied in order for the specified permissions to be granted. The boolean expressions can be based on the user identity, role, or type attributes in the pair of security contexts.

Example 3.18. SELinux - Constraint definitions

```
constrain process transition ( u1 == u2 or t1 == privuser );
constrain process transition ( r1 == r2 or t1 == privrole );
constrain dir_file_class_set { create relabelto relabelfrom }
( u1 == u2 or t1 == privowner );
```

3.4.5. Security Context Specifications

The security contexts specifications provide security contexts for various entities such as initial SIDs, pseudo file-system entries, and network objects. It also specifies the labelling behaviour to use for each file-system type.

Initial SIDs are SID values that are reserved for system initialisation or predefined objects. The initial SID contexts configuration specifies a security context for each initial SID. A security context consists of a user identity, a role, and a type.

Example 3.19. SELinux - Initial SID Contexts

```
sid kernel system_u:system_r:kernel_t
sid init   system_u:system_r:init_t
sid proc   system_u:object_r:proc_t
```

Chapter 4. Proposed XML grammar

4.1. Properties of proposed XML grammar

The XML description of RBAC and DTE model has been proposed with regard to:

- Expressiveness
- Extensibility
- Easy portability
- Flexibility

After detailed analysis of existing grammars describing the RBAC and DTE model, we decided to propose an XML [XML] structure which fully covers the description of the RBAC and DTE models written using existing grammars. The most exhaustive description of RBAC can be found in the RSBAC implementation, the description of DTE in the SELinux implementation. The XML structure has been proposed regards to these facts.

The strong expressiveness of XML allows to define mandatory and optional elements, their mandatory and optional attributes with default values etc. Due to this XML feature, it is possible to fill up the XML structure partially and use only certain values of elements or attributes during the appropriate transformation. Therefore, the XML description remains extensible to new security models, new properties of existing security models and various particularities of concrete security project (rules for their generation have to be included in an appropriate transformation style-sheet).

The proposed XML structure involves a declaration of:

- Security classes (Flask)
- Types (domains) and attributes (DTE)
- DTE access matrix, transition rules, change rules (DTE)
- Constraints (Flask)
- Initial SIDs (Flask)
- Parent, child, compatible, privileged roles (RBAC)
- Users and their access rights (DAC+RBAC)

4.2. Security classes (Flask)

For a more fine-grained description of security permission, the Flask architecture introduces the so-called Flask security classes. Each permission is associated with a security class. Each security class defines a set of allowed permissions that can be associated with. In the SELinux implementation a permission involves a basic kernel syscall, an extended LSM kernel syscall or a special SELinux-defined (transition) function. In a generic XML description anything can be supposed to be a permission, which will be recognised during the XSL transformation. The names of security classes are chosen as the names of the system entities (tcp/udp socket, file, directory, IPC message etc.) A lot of security frameworks do not use security classes. For these frameworks it is better to use a set of permissions aggregated from the permissions in all the security classes. We have to take into account that any aggregation causes a loss of security information, consequently any aggregation used in a transformation will produce a security-inferior configuration policy.

Example 4.1. Proposed XML grammar - security classes (DTD)

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT classes (class+)>
<!ATTLIST class name (security|process|system|capability|filesystem|file|
dir|fd|lnk_file|chr_file|blk_file|sock_file|fifo_file|socket|tcp_socket|
udp_socket|rawip_socket|node|netif|netlink_socket|packet_socket|key_socket|
unix_stream_socket|unix_dgram_socket|sem|msg|msgq|shm|ipc|passwd) #IMPLIED>
<!ELEMENT class (pname+)>
<!ELEMENT pname (#PCDATA)>
```

4.3. Types (domains) and attributes declaration (DTE)

A types and attributes declaration is required in every implementation of the DTE model. Similarly, as the TE model simplifies a domain to a type with the *domain* attribute, the proposed XML structure also considers a domain to be a type with a *domain* attribute. This simplification has no impact on the security characteristic of subject to object relations because the state of the DTE access matrix, transition rules and change rules remain unchanged. The type is represented by the XML element **type**, where an attribute *name* defines its name, subelements **attribute** its attributes. The attributes describe generic properties of types. Therefore, it is possible to define access vectors, transition or change rules for types with common attributes. Further subelements are system-specific as **filename**, **fsuse**, **port**, **interface**, **node**, which associate the type with real files (or directory structure expressed by a regular expression), a block or character device, a filesystem type, TCP/UDP ports, a network interface or network node.

Example 4.2. Proposed XML grammar - type declaration (DTD)

```
<?xml version="1.0" encoding="UTF-8"?>

<!ELEMENT types (type+)>
<!ELEMENT type (attribute*, (filename*|fsuse*|port*|interface*|node*))>
<!ATTLIST type name NMTOKEN #REQUIRED>
<!ELEMENT filename (#PCDATA)>
<!ATTLIST filename fs NMTOKEN #IMPLIED>
<!ATTLIST filename attr (file|directory|character) #IMPLIED>
<!ELEMENT fsuse (#PCDATA)>
<!ATTLIST fsuse name NMTOKEN #REQUIRED>
<!ELEMENT port (#PCDATA)>
<!ATTLIST port protocol (tcp|udp) #IMPLIED>
<!ELEMENT interface (#PCDATA)>
<!ATTLIST interface default_msg_type NMTOKEN #IMPLIED>
<!ELEMENT node (address,netmask)>
<!ELEMENT address (#PCDATA)>
<!ELEMENT netmask (#PCDATA)>
<!ELEMENT attribute (#PCDATA)>
```

4.4. DTE access matrix, transition rules and change rules declaration (DTE)

The proposed XML structure describes the following DTE components:

- Access Vectors rules (AV rules) of the DTE matrix
- Transition rules
- Change rules

All DTE relations for each type to all other are embedded in a **domain** element. The above-mentioned rules hold for each type, thus are defined as subelements of the **domain** element. They express a relation of a given type (domain) to each other and all possible transitions/changes from given type (domain) to another type (domain).

Example 4.3. Proposed XML grammar - DTE definition (DTD)

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT dte (domain+)>
<!ELEMENT domain (avrules+,transitions,changerules)>
<!ATTLIST domain name NMTOKEN #REQUIRED
attributes NMTOKENS #IMPLIED>

<!ELEMENT avrules (avtype+)>
<!ELEMENT transitions (transtype*)>
<!ELEMENT changerules (changetype*)>

<!ELEMENT avtype (permission+)>
<!ATTLIST avtype name NMTOKEN #REQUIRED
audit (yes|no|none) "none"
class NMTOKEN #REQUIRED>

<!ELEMENT transtype (#PCDATA)>
<!ATTLIST transtype name NMTOKEN #REQUIRED
class NMTOKEN #REQUIRED>

<!ELEMENT changetype (#PCDATA)>
<!ATTLIST changetype name NMTOKEN #REQUIRED
class NMTOKEN #REQUIRED>

<!ELEMENT permission (#PCDATA)>
```

4.4.1. Access Vector rules (AV rules) of DTE matrix

All access vectors related to a given domain are embedded in a **avrules** element. An **avtype** element describes exactly one access vector. Its attribute *name* represents the name of the target type. A type which is referred by the *name* attribute of the parent **domain** element is considered to be the source type. The next attribute of **avtype** is a *class* representing the security class usable only in the Flask architecture. Further subelements involve a set of allowed permissions between the source and target type. It is also possible to define an optional attribute of **avtype** called *audit*. This attribute has no control meaning - it defines whether to audit or not occurred events between types. It makes sense only for security policy debugging purposes.

4.4.2. Transition rules

All transition rules related to a given domain are embedded in a **transitions** element. A **transtype** element describes exactly one transition rule. Its *name* attribute represents a name of the target type. A type which is referred by the *name* attribute of the parent **domain** element is considered to be the source type. The element **transtype** defines

1. A new type of object created by the process of the source type that accesses the object of the target type (e.g. the parent directory for files).

Example 4.4. Proposed XML grammar - TE transition rules (object transition)

```
<domain name="sshd_t">
<transitions>
<transtype name="tmp_t" class="file">sshd_tmp_t</transtype>
<transtype name="tmp_t" class="dir_file">sshd_tmp_t</transtype>
<transtype name="tmp_t" class="lnk_file">sshd_tmp_t</transtype>
<transtype name="tmp_t" class="sock_file">sshd_tmp_t</transtype>
<transtype name="tmp_t" class="fifo_file">sshd_tmp_t</transtype>
</transitions>
</domain>
```

2. A new type for the process domain, in case the domain with the source type is referred to the object (e.g. executable) of the target type.

Example 4.5. Proposed XML grammar - TE transition rules (subject transition)

```
<domain name="sshd_t">
<transitions>
<transtype name="shell_exec_t" class="process">user_t</transtype>
</transitions>
</domain>
```

4.4.3. Change rules

All change rules related to a given domain are embedded in a **changerules** element. A **changetype** element describes exactly one change rule. Its attribute *name* represents a target type name. A type which is referred by the *name* attribute of the parent **domain** element is considered to be the source type. The change rules can be useful in security-aware (LSM) applications, i.e. applications that are able to change the type of the objects they generate (e.g. the type of character devices `/dev/pts*` will be changed after user login to the system by the security-aware (LSM) application `login`, `ssh` etc.) The objects that are generated by ordinary (not security-aware) application can have only one type in the scope of one source and target type[11], which is given by the corresponding transition rule.

[11]This type is given by the new type of the above-mentioned TE object transition.

Example 4.6. Proposed XML grammar - TE change rules

```
<domain name="sysadm_t">
<changerules>
<changetype name="sysadm_tty_device_t" class="chr_file">sysadm_tty_device_t
</changetype>
<changetype name="sshd_devpts_t" class="chr_file">sysadm_devpts_t
</changetype>
</changerules>
</domain>
```

4.5. Constraints declaration

Constraints make possible to define conditions that have to be fulfilled during the generation of a valid security policy configuration. They ensure policy consistence and disallow the use of security rules conflicting with them (may be useful in case of automatic generated rules). **permission** subelements together with an attribute *class* of the **constraint** element describe a permission for the fulfilled algebraic condition given by the **expr** element. Due to the fact that the algebraic condition can contain further recursive subexpressions, we decided to represent the constraints in XML using the reverse polish notation.

Example 4.7. Proposed XML grammar - Constraints definition (DTD)

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT constraints (constraint+)>
<!ELEMENT constraint (permission+, expr+)>
<!ATTLIST constraint class NMTOKEN #REQUIRED>
<!ELEMENT expr (#PCDATA | expr)*>
<!ATTLIST expr operator (or|not|and|eq|neq|dom|domby|incomp) #IMPLIED>
```

The following constraint prevents role changes unless the current domain of the process is in the set of types with the *privrole* attribute.

Example 4.8. Proposed XML grammar - Constraints definition

```
<constraint class="process">
<permission>transition</permission>
<expr operator="or">
<expr operator="eq">
<expr>t1</expr>
<expr>privrole</expr>
</expr>
<expr operator="eq">
<expr>r1</expr>
<expr>r2</expr>
</expr>
</expr>
</constraint>
```

4.6. Initial SIDs declaration (Flask)

This declaration is related only to the Flask architecture. It describes initial values of secure identifiers.

Example 4.9. Proposed XML grammar - Initial SIDs (DTD)

```
<?xml version="1.0" encoding="UTF-8"?>

<!ELEMENT initialsids (sid+)>
<!ELEMENT sid (#PCDATA)>
<!ATTLIST sid name NMTOKEN #REQUIRED>
```

4.7. Parent, child, compatible, privileged roles declaration (RBAC)

The XML description of RBAC has been proposed regarding to existing RBAC implementations [RS-BACImpl] and [SELinuxImpl]. Role declarations are embedded in a **roles** element. Each role can optionally contain a set of parent roles **parentroles**, child roles **childroles**, compatible roles (RSBAC) **compatibler-oles**, one admin role and a set of associated types **rtypes**. The DTD does not ensure that the defined child and parent roles are consistent (e.g. if role A contains a child B, then the writer of the policy must specify A as the parent of B). It is possible to define various sets of types depending on the *type* (file, device, process, ipc etc.) attribute of the **rtypes** element. The *type* has a different meaning with the **rtype** element. In this case it defines a default type that will be used with the creation of new files, processes, IPC etc. for given role [12].

[12]The default type is used in RSBAC implementation

Example 4.10. Proposed XML grammar - RBAC definition (DTD)

```
<?xml version="1.0" encoding="UTF-8"?>

<!ELEMENT rbac (role+)>
<!ELEMENT role (parentroles*,childroles*,adminrole?,compatibleroles*,
rtypes*)>
<!ATTLIST role name NMTOKEN #REQUIRED>
<!ELEMENT parentroles (parentrole*)>
<!ELEMENT childroles (childrole*)>
<!ELEMENT adminrole (#PCDATA)>
<!ELEMENT parentrole (#PCDATA)>
<!ELEMENT childrole (#PCDATA)>
<!ELEMENT compatibleroles (crole*)>
<!ELEMENT crole (#PCDATA)>
<!ELEMENT rtypes (rtype+)>
<!-- RSBAC types -->
<!ATTLIST rtypes type (fd|dev|process|ipc|scd|none) #IMPLIED>
<!ELEMENT rtype (#PCDATA)>
<!ATTLIST rtype type (fd_create_type|process_create_type|process_chown_type
|process_execute_type|ipc_create_type|none) #IMPLIED>
```

4.8. Users and their access rights declaration (DAC+RBAC)

User declarations are embedded in a **users** element, each user can be associated with one or many roles. Permissions are not associated directly with the users, they are associated with corresponding types or roles. This concept is preferred by most security projects. It implies better robustness and extensibility.

Example 4.11. User/subject assignment (DTD)

```
<?xml version="1.0" encoding="UTF-8"?>

<!ELEMENT users (user+)>
<!ELEMENT user (urole+)>
<!ATTLIST user name NMTOKEN #REQUIRED>
<!ELEMENT urole (#PCDATA)>
```

4.9. Proposed XML grammar generator

For demonstration purposes we implemented an XML generator of the proposed grammar. We chose SELinux as a security framework since it implements both the RBAC and TE security model. We consider 2 methods of processing the input security policy:

1. To propose and implement a proper parser of the SELinux policy with the ability to generate an XML description.

2. Let the existing SELinux parser (`checkpolicy`) with added functionality to parse SELinux policy and then generate the XML description from valid and correct memory structures.

The first method involves the creation of a relative complex parser able to perform lexical analysis, expand m4 macros and apply LALR decomposition recursively, etc. Also maintaining this parser would mean a lot of work because the SELinux developers frequently add or change the policy format. The second method is much simpler, the existing SELinux parser validates and checks the policy, then maps a binary representation (all macros are already expanded) of the policy to the memory. Then it is quite easy to interpret this binary structure and generate an XML representation.

Due to the above-mentioned reasons we decided to choose the second method. We wrote an XML generator which transforms the SELinux policy mapped in the memory to the desired XML representation. The generator was implemented in C language. After executing a modified version of `checkpolicy` the following files will be generated: `classes.xml`, `constraints.xml`, `dte.xml`, `initialsids.xml`, `rbac.xml`, `types.xml` and `users.xml`.

The declaration of the SID for file objects (a combination of a SELinux user, role and type) is not stored in a global SELinux policy `policy.conf`, but directly in extended ext3 file-system attributes. These declarations are placed in file context descriptions located in `/etc/selinux/file_contexts/` and set by the SELinux command `setfiles`. We wrote a short script `generate_fc.sh`, which parses these declaration and generates the appropriate XML description with the `types.dtd` document type definition. The script implements the following operations:

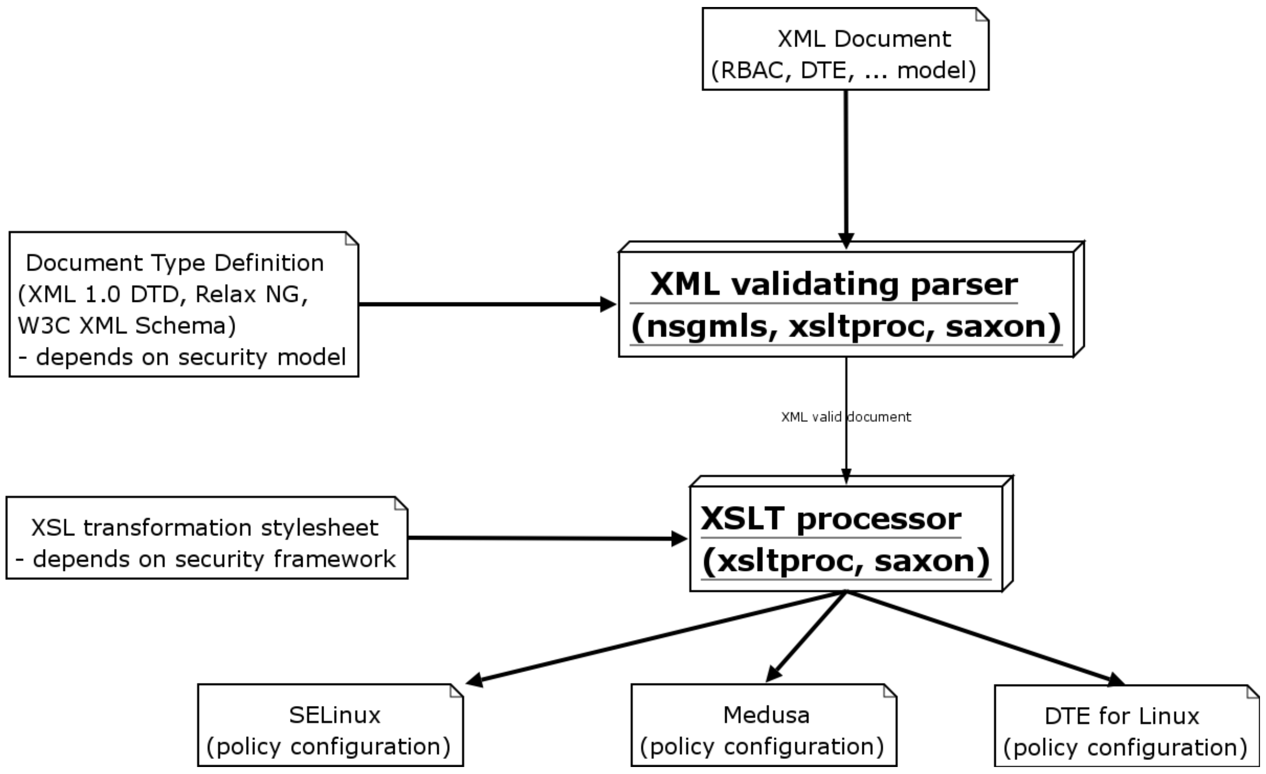
- Grouping and sorting of each type used in the file contexts descriptions.
- Detection of a regular expressions and consequently recognition of directories with an appropriate setting of the *attr* attribute in a **filename** output element.
- Decomposition of one regular expression, which matches a full pathname, to many partial regular expressions according to the separator "/" (a faster and simpler representation)[13]

[13]This decomposition solves the incompatibility problems between SELinux, where regular expressions are referred to full pathname and Medusa, where the full pathname is a composition of many regular expressions.

Chapter 5. Proposed transformations

There are a lot of existing style-sheet languages designed for XML transformation. The most famous and popular are Document Style Semantics and Specification Language (DSSSL) [DSSSL] proposed by James Clark and The Extensible Style-sheet Language [XSL] proposed by W3C. After an analysing the above-mentioned languages and existing transformation processors we decided to use the eXtensible Style-sheet Language (XSL). The DSSSL is a language more suitable for publication purposes [DSSSLforXML]. Open source XSL processors have been used for the transformation - the fast xsltproc and the robust saxon.

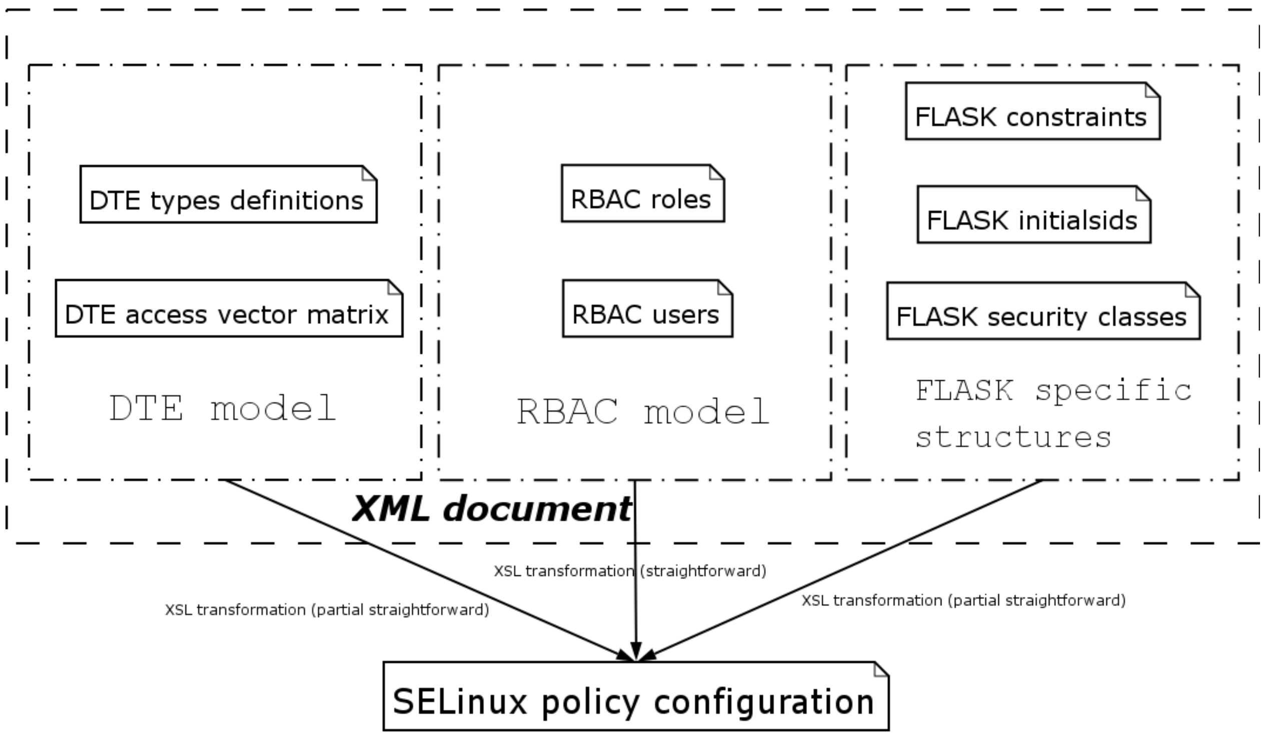
Figure 5.1. Proposed transformation of security models



5.1. SELinux

Due to the fact that the XML structure was based on the concept of SELinux, the transformation of the XML description to a SELinux policy is relatively easy.

Figure 5.2. Proposed transformation of the RBAC+DTE model and Flask specific structures to a SELinux policy configuration


DTE types definitions
DTE access vector matrix
DTE model
RBAC roles
RBAC users
RBAC model
FLASK constraints
FLASK initialsids
FLASK security classes
FLASK specific structures
XML document
XSL transformation (partial straightforward)
XSL transformation (straightforward)
XSL transformation (partial straightforward)
SELinux policy configuration

5.1.1. Types transformation

The SELinux grammar requires a declaration for any type and any used attribute. This information is not included directly in the XML representation, but declared as an optional *attribute* attribute of the defined **type** element. Therefore, the transformation of type's attributes requires a retrieval of all unique usages of each attribute. The result of the retrieval (a list of all unique attributes) represents the declaration of all the attributes. The proposed transformation considers the fact that the declaration of a type can be written in the SELinux policy in various ways (as a file, interface, network interface, network node, filesystem, etc.).

5.1.2. Transformation of DTE access matrix

The SELinux policy has to be easy modifiable in an user-friendly way. An user should be able to insert his own rules to the SELinux policy in any order. This property is achieved by implicit addition of the source and target type to the SELinux policy for every rule. On the contrary, in an XML representation, the information about the DTE access matrix is structured in a hierarchical manner. It follows the source type definition appears in XML only once and all the relations from this type to each other are located in its child elements. This representation of types is shorter and more suitable for any transformation. The transformation is applied in successive steps for each source type. For each relation defined in the child elements of a given source type and target type, the access vector, transition and change rule in the SELinux grammar are generated. In the case of the generated access vectors (AV) rules we consider an optional *audit* attribute, which defines whether to audit the events which occur between types. Values of "yes" or "no" of this attribute have an audit meaning, whereas the "none" value has a control meaning. The transformation of the transition and change rules is analogical to the above-mentioned transformation of access vector (AV) rules with the source type as a *name* attribute of the **domain** element.

5.1.3. Transformation of RBAC roles

An XML representation of RBAC involves all RBAC attributes of SELinux. This implies the transformation of RBAC to a SELinux policy is straightforward (i.e. each role is associated with a set of allowed types). Other attributes of RBAC remain unused in the SELinux transformation.

5.1.4. Transformation of (RBAC) users

The SELinux grammar has the same representation of the users as the proposed XML grammar. It follows the transformation of the users is straightforward (i.e. each user is associated with a set of allowed roles).

5.1.5. Transformation of constraints

The transformation of the constraints requires recursively calling the transformation template which evaluates a nested **expr** element. This element contains the resulting algebraic condition represented in reverse polish notation. The XSL template has to be called once for the operator "not" or twice for the rest of operators with two operands. When no operator is used, the recursive calling will terminate.

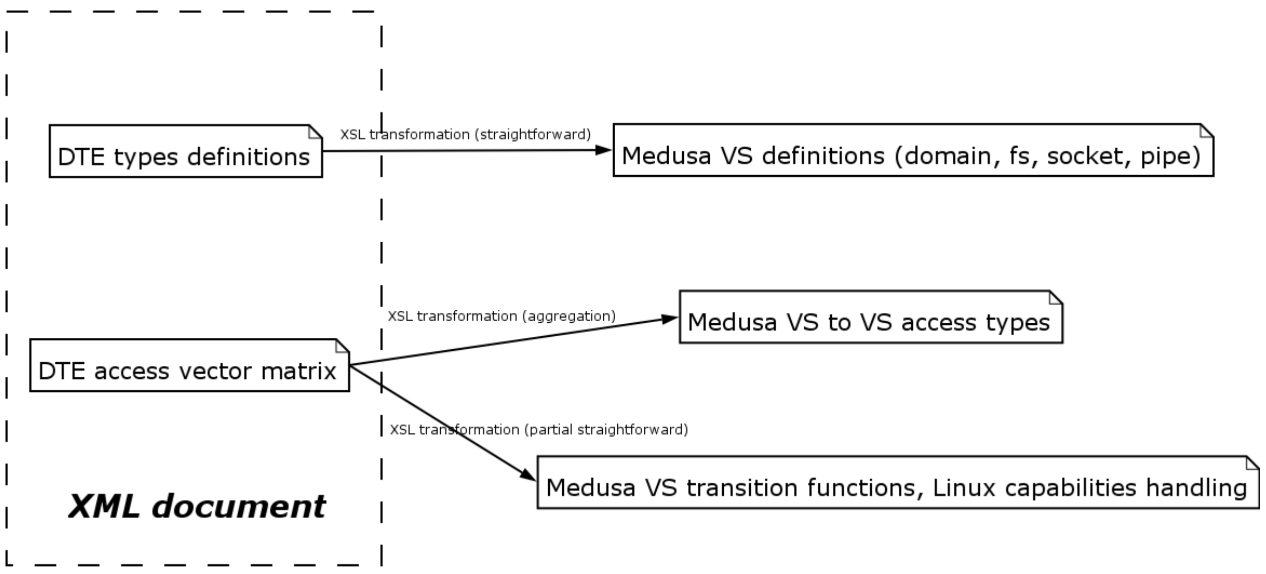
5.1.6. Transformation of initial SIDs and security classes

The transformation of the security classes to a SELinux policy is straightforward. The XSL style also appends a declaration of initial SIDs to the resulting policy.

5.2. Medusa

Medusa is a ZP security framework implementation [ZelemPikula2001], which introduces VS (virtual spaces) and defines subject and object association with VS and relations between each VS. The concept of VS has a few similar aspects with the DTE model, therefore we decided to take advantage of this analogy during the design of the transformation. The tested version of Medusa (1.0 for 2.4.25 Linux kernel) doesn't implement the RBAC model yet which was the reason we didn't implement a RBAC transformation. Also the tested version of Medusa that has been used doesn't yet implement support of object as tcp/udp ports, network interfaces, network nodes, type of filesystem etc. Therefore the DTE access vector or transition rules that contain this kind of types can not be transformed to Medusa. Due to the fact that the Medusa doesn't directly support the DTE model, the transition mechanism between VS is a bit different than the transition mechanism between DTE types and domains. This difference between ZP security framework's and DTE's implementation makes a full transformation impossible, so we proposed and implemented only partial transformations of the DTE model.

Figure 5.3. Proposed transformation of the DTE model to Medusa VS


DTE types definitions
XSL transformation (straightforward)
Medusa VS definitions (domain, fs, socket, pipe)
XSL transformation (aggregation)
Medusa VS to VS access types
DTE access vector matrix
XSL transformation (partial straightforward)
Medusa VS transition functions, Linux capabilities handling
XML document

5.2.1. Transformation of DTE types to VS

This transformation involves a transformation of the types from the XML representation `types.xml` and `exttypes.xml`[14]. We consider two ways of transformation - the first one when a type is associated with a file and the second one when a type is associated with a socket, process, port, network device etc.

- Medusa - file type transformation

  A VS declaration is generated for each **type** element. If the *attr* attribute of the **filename** element is "directory" a *recursive* keyword is used, in other cases not.

Example 5.1. Medusa - file type transformation

```
space amavisd_etc_t             = "/etc/amavisd.conf";
space amavisd_exec_t            = "/usr/sbin/amavisd.*";
space amavisd_lib_t             = recursive "/var/lib/amavis";
space amavisd_log_t             = "/var/log/amavisd.log";
space amavisd_var_run_t          = recursive "/var/run/amavis";
```

[14]The separation of type definition to these two files has only an informative meaning. The `types.xml` contains type definitions obtained from the SELinux security policy configuration, the `exttypes.xml` contains type definitions located directly in the extended attributes of filesystem. Both these files have the same DTD definition, therefore we are planning to merge these definition to one file `types.xml`.

- Medusa - non-file type transformation

  We consider a transformation of type associated with:

  - **domain of process**

    The *primary space* keyword is used with a *domain* node in the transformation output.

    Example 5.2. Medusa - domain type transformation

    ```
    primary space ping_t = "domain/ping_t";
    ```

  - **socket**

    A *socket* node is used in the transformation output.

    Example 5.3. Medusa - socket type transformation

    ```
    space icmp_socket_t = recursive "/socket:";
    ```

  - **pipe**

    A *pipe* node is used in the transformation output.

    Example 5.4. Medusa - pipe type transformation

    ```
    space pipe_t = recursive "/pipe:";
    ```

  - **type of filesystem, network interface, tcp/udp port, network node**

    Due to the fact that Medusa does not have modules for VS representation of these objects, a trivial transformation is not possible. Therefore the resulting transformation output is not functional yet.

    Example 5.5. Medusa - network interface, fs, port, network node type transformation

    ```
    space netif_eth0_t = recursive "/network/eth0";
    space telnet_port_t = recursive "/port/tcp/23";
    space iso9660_t =                recursive "/iso9660/";
    ```

5.2.2. Transformation of DTE access matrix to VS access types

The transformation of the DTE access matrix to VS access types is the most complex and time consuming transformation. The transformation is also lossy because of the strong aggregation of permissions between types and domains. It implies a degraded usability of the resulting transformation. We proposed several transformation style-sheets with various transformation times [15]. The last and fastest version of transformation is divided into the following parts:

1. For each domain in the DTE access matrix, all types related to this domain are classified into four sets regarding to the type of access (READ, WRITE, SEE, ENTER). This classification is done according to the type of permission between each domain's type and this domain. The permission between types and domains is expressed by the security class and the **permission** element. Due to the fact that type access between VS can be classified only as reading, writing, seeing and entering, the security class is ignored during this classification. Thus the classification depends on the **permission** element. This is the first part of transformation. The resulting XML file contains all the domain declarations and each domain contains four sets of reading (**reads**), writing (**writes**), seeing (**sees**) and entering (**enters**) permissions. The classification is applied only for control meaning **avtype** elements, i.e. the elements which do not have set an *audit* attribute.

2. For each classified set of the domain (i.e. **reads**, **writes**, **sees**, **enters**) the duplicated types are eliminated. Simultaneously, a number of types for each set is kept - this information is used for ensuring the generation of a syntactically correct Medusa policy. The elimination of the duplications is implemented using XSL `key` and the XSL function `generate-id`.

The degraded aggregation usability is caused by

- Ambiguity of the classification - a problematic categorisation of XML permission to a given set (**reads**, **writes**, **sees**, **enters**)
- Absence of Medusa modules needed for VS representation of the following objects - tcp/udp port, network node, network interface, etc. - the rules in the XML representation that contain such types of object are transformed to invalid (but unused) rules in Medusa
- Unsupported permissions - permissions that are represented by the LSM syscall do not work in Medusa
- Insufficient DTE transformation - Medusa does not have direct support of the DTE model, only a few aspects of the DTE model can be directly transformed, other ones need an additional hack to constable, e.g. transition rules need to use the `fexec:NOTIFY_ALLOW` and `enter` functions between subject and object.

[15]The first version of DTE transformation from generic XML policy took a long time (4 hours).

Example 5.6. DTE access vector to VS access - aggregation

```
<xsl:element name="reads">
<xsl:apply-templates  select="descendant::permission[.='bind' or
.='ioctl' or .='execute' or .='execute_no_trans' or .='rawip_recv'
or .='udp_recv' or .='tcp_recv' or .='read' or .='receive'
or .='recvfrom' or .='recv_msg' or .='syslog_read' or .='unix_read']"/>
</xsl:element>

<xsl:element name="writes">
<xsl:apply-templates select="descendant::permission[.='append' or
.='connect' or .='connecto' or .='create' or .='write'
or .='relabelfrom' or .='relabelto' or .='destroy' or .='enqueue'
or .='chfn' or .='passwd' or .='chsh' or .='ioctl' or .='link'
or .='unlink' or .='listen' or .='lock'  or .='mount' or .='remount'
or .='mounton' or .='unmount' or .='name_bind' or .='newconn'
or .='quotamod' or .='quotaon' or .='send' or .='send_msg' or .='sendto'
or .='tcp_send' or .='udp_send' or .='remove_name' or .='rename'
or .='getattr' or .='setcap' or .='setexec' or .='setfscreate'
or .='setopt' or .='setpgid' or .='setrlimit' or .='setsched'
or .='shutdown' or .='swapon' or .='syslog_mod' or .='unix_write'
or .='write' or .='rawip_send']"/>
</xsl:element>

<xsl:element name="sees">
<xsl:apply-templates select="descendant::permission[.='ioctl'
or .='getattr' or .='getcap' or .='getopt' or .='getpgid'
or .='getsession' or .='getsched' or .='ipc_info' or .='quotaget'
or .='search' ]"/>
</xsl:element>

<xsl:element name="enters">
<xsl:apply-templates select="descendant::permission[.='entrypoint'
or .='transition']"/>
</xsl:element>
```

5.2.3. Medusa capability handling according to the DTE access matrix

The capability handling in Medusa can be implemented in two ways:

- By hooking a syscall `exec` using the `fexec` constable function. Then it is possible to set Linux capabilities in a particular hooking procedure.

Example 5.7. Medusa capability handling through `fexec/sexec`

```
subject_vs fexec object_vs {
..
..
process.ecap = CAP_NET_RAW;
process.pcap = CAP_NET_RAW;
process.icap = CAP_NET_RAW;
..
..

}
```

- By emulation of POSIX file capabilities. A constable function `getfile` allows to define a relevant file capability for a selected file. In this case the setting of Linux capabilities in the `fexec` hooking procedure does not work.

Example 5.8. Medusa capability handling in the case of the file capability emulation

```
"/usr/bin/traceroute"   getfile:NOTIFY_ALLOW * { pcap=CAP_NET_RAW; }
"/bin/su"               getfile:NOTIFY_ALLOW * { pcap=CAP_SETUID+
CAP_SETGID; }
```

In the first case the Linux capabilities are set when the subject (process) starts in a corresponding domain VS. On the contrary, in the second case, the capabilities are the properties of a file on a (simulated file capability) filesystem. The file can be associated with a different VS as a process created from running this file. We implemented a XSL transformation for both cases. The Linux capabilities are involved in the DTE access matrix as permissions of a domain (a type with the *domain* attribute) related to selfsame domain (using *self:class*). In the case of the file capability emulation, we have to scan all the types associated with file objects and for each type, look for an allowed transition to a domain (look for "entrypoint" permission), which has defined Linux capabilities (using *self:class*). The capabilities found this way correlate with the given files. This method is too complicated as well as inaccurate due to the fact that there are a lot of unrelated transitions between the type of the file and domain with defined Linux capabilities. These unrelated transitions cause the generation of unused capabilities in the resulting Medusa policy. That was reason we decided to use a better and smarter solution with setting capabilities in the hooked `exec`. The `fexec`, respectively `sexec` constable function allows hooking of kernel syscall and adding, removing or setting appropriate Linux capabilities in our customised hooked procedure. A transition to another VS can be also implemented in the hooked procedure. We wrote a transformation style-sheet which generates these procedures for all the processes that needed Linux capabilities or transitions to another VS. The constable allows to define a source VS of the subject for the `fexec` and `sexec` functions. The transition to the target VS is allowed only for subjects associated with this VS. We find the list of all source VS by scanning the DTE matrix and looking for the types with allowed permission `execute` (needed for executing files) to the target VS. An union of all these types for the given target VS represents a list of the source VS.

Example 5.9. Generation of Medusa transition functions with capabilities

```
space traceroute_exec_t         = "/bin/traceroute.*"
+ "/usr/(s)?bin/traceroute.*"
+ "/usr/bin/lft"
+ "/usr/bin/nmap"
+ "/bin/traceroute.*"
+ "/usr/(s)?bin/traceroute.*"
+ "/usr/bin/lft"
+ "/usr/bin/nmap";

primary space traceroute_t = "domain/traceroute_t";

space system_crond_t_sysadm_t_staff_ssh_t_staff_crond_t_traceroute_t\
_sysadm_ssh_t_sysadm_crond_t_user_t_user_ssh_t_user_crond_t_staff_t\
_initrc_t = space system_crond_t + space sysadm_t + space staff_ssh_t
+ space staff_crond_t + space traceroute_t + space sysadm_ssh_t +
space sysadm_crond_t + space user_t + space user_ssh_t +
space user_crond_t + space staff_t + space initrc_t;

system_crond_t_sysadm_t_staff_ssh_t_staff_crond_t_traceroute_t\
_sysadm_ssh_t_sysadm_crond_t_user_t_user_ssh_t_user_crond_t_staff_t\
_initrc_t fexec:NOTIFY_ALLOW traceroute_exec_t {

process.ecap = CAP_SETGID + CAP_SETUID + CAP_NET_ADMIN + CAP_NET_RAW;
process.pcap = CAP_SETGID + CAP_SETUID + CAP_NET_ADMIN + CAP_NET_RAW;
process.icap = CAP_SETGID + CAP_SETUID + CAP_NET_ADMIN + CAP_NET_RAW;

enter_domain ("traceroute_t");
log_proc ("traceroute_t");
}
```

5.3. DTE for Linux

Due to the absence of a documentation of DTE for Linux [DTE-for-Linux2000] and source code of DTE kernel modules (there are only binary versions for old kernel version) we decided not to implement XSL transformation for this implementation. Relations between domains and types are convertible to relations between VS in Medusa implementation (e.g. read, write, execute type of access). Therefore, the transformation requires an aggregation of type permissions used in the XML representation of the DTE access matrix. The type declaration (declared using the **assign** keyword in the DTE for Linux policy) is transferable from the XML representation of the types in a straightforward manner. Due to the fact that the development of DTE for Linux is stalled (last changes were in the 2001), and its full functionality is covered by SELinux [SELinuxImpl], the future development of transformation styles makes no sense.

5.4. Implementation status

Table 5.1. Transformation style-sheets - implementation status

	SELinux		**Medusa**	
Tested environment	Patched SELinux kernel 2.6.3		Patched Medusa kernel 2.4.25	
Model property	Implemented transformation	SELinux native support	Implemented transformation	Medusa native support
Security classes	Yes	Yes	No	No
Constraints	Yes	Yes	No	No (a need to use handling function)
Initial SIDs	Yes	Yes	No	No
RBAC	Yes	Yes	No	No (does not work in current version)
DTE types	Yes	Yes	Yes	No (a mapping to VS is used)
DTE matrix	Yes	Yes	Partial	No (a mapping to access type between VS is used)

Security classes, constraints and initial SIDs are properties of the Flask architecture which is implemented only in the SELinux. Constraints can be implemented in Medusa using a strong Constable language. RBAC model was supported in previous versions of Medusa, but there is no support in the current version. Medusa does not support directly any security model, but can be a framework (through virtual spaces) for its implementation. DTE types can be directly mapped to VS and a DTE matrix can be partially mapped to access types between VS. SELinux uses capabilities from the DTE matrix, Medusa needs a special generated handling function, which sets correct Linux capabilities. A RSBAC security policy is configured using a set of user-space utilities. RSBAC does not use a centralised plain-text configuration, but it is possible to generate a script able to create the RSBAC security policy through the XSL style-sheet.

Chapter 6. Conclusion

The thesis analysed possible methods for representation of RBAC and DTE access control models. Also other well-known computer security models and their implementations have been studied. It has been shown that an XML representation, thanks to its properties, satisfies most of requirements for extensible and easily transformable representation. The DTD for RBAC and DTE security models has been proposed and a sample XML structure has been filled. The SELinux and Medusa demonstration transformation style-sheets have been written and applied to the sample XML representation. A lot of transformation issues have been discovered and possible solutions have been proposed. This thesis also provides us a new perspective on existing security implementations described in the State-flow security model extension (SFSME).

6.1. Future use

6.1.1. Integration of the XML security configuration policy to Linux distributions

The integration of the XML security configuration policy involves the creation of a partial XML configuration for each application (daemon) that needs a special security protection. Suitable properties of the XML representation of the RBAC and DTE security models allows to easily generate custom XML configurations for a given daemon. This XML configuration can be inserted into an appropriate distribution's package (deb, rpm, pkg, etc). A proper post-install script will detect the type of security framework (Medusa, SELinux, etc) and apply an appropriate XSL style to the given XML configuration. The XSL style can be installed globally or locally - in the case of a specially customised generated output. The result of the transformation will be a part of the syntactically and lexically correct configuration policy according to the given grammar of a security project's validator (constable, checkpolicy, etc). This part of the configuration policy can be appended to the main configuration of the security project's policy (in the case of SELinux it will be necessary to copy the policy to `/etc/selinux/domains/program/` and remake the policy using **make policy**. In case of Medusa it will be necessary to insert a **#include** statement to a global configuration file with an appropriate correct reference). The uninstallation of the installed package with the generated XML configuration will be a trivial operation - deleting or moving the generated part, fixing the created references and remaking the security policy. The integration of the XML policy into the software packages will simplify the administration of a system based on a strong security framework. The packages modified in the described way can be used with no or minimal changes on various supported security frameworks.

6.1.2. Portability of XML security policy to another Unix OS

Since the resource types of security system - subjects (processes) and objects (files, sockets, IPC messages, etc.) are common for all Unix OS, main differences lie in various permission types (mediating through various syscalls) of each Unix OS. Hence a successful transformation of the XML security policy from one Unix OS to another requires a suitable way to map of permission types (syscall calls) between each OS. The mapping can be an aggregation if the target kernel has a reduced amount of syscalls or a duplication of permission types if the target kernel has a greater number of syscalls than the source kernel. By using this method it is possible to transform e.g. the SELinux policy to *BSD systems (SEBSD project in TrustedBSD), Solaris etc.

6.1.3. Extensibility of XML representation

An XML representation is easily extensible to describe new security models or new attributes of existing security models without impact on functionality. It is also possible to add attributes applicable only in one type of specific transformation (causing a loss of interoperability and portability), e.g. attributes of Flask architectures etc. Modifying the XSL style-sheets makes it possible to create a security policy for new security projects or extends the security policy of existing security projects. Developers of XSL transformations should distinguish between security model's properties (and implement them in XML representation) and specific implementation's properties (and implement them in XSL style-sheet). This separation provides a cleaner and well-arranged XML policy.

Bibliography

[AmonOtt2002] Amon Ott. *The Role Compatibility Security Model*. http://www.rsbac.org/rc-nord-sec2002.pdf.

[Boebert1985] W.E. Boebert and R.Y. Kain. *A practical Alternative to Hierarchical Integrity Policies*. Proceedings of the 8th National Computer Security Conference. September 1985.

[ClarkWilson1997] D.D. Clark and D.R. Wilson. *Non Discretionary Controls Commercial Applications*.

[DTEconfining] Lee Badger, Daniel F. Sterne, Michael J. Petkac, Kenneth M. Walker, David L. Shermann, and Karen A. Oostendorp. *Confining Root Programs with Domain and Type Enforcement (DTE)*. http://www.usenix.org/publications/library/proceedings/sec96/full_papers/walker/walker.ps . July 1996.

[DACdefinition] Copyright © 2001 American National Standard for Telecommunications. http://www.atis.org/tg2k/_discretionary_access_control.html.

[DTE-for-Linux2000] Serge E. Hallyn and Phil Kearns. *Domain and Type Enforcement for Linux*. College of William and Mary. http://www.cs.wm.edu/~kearns/001lab.d/projects.d/als2000.pdfImplementation - http://www.cs.wm.edu/~hallyn/dte/ .

[DTE-Unix-Prototype1995] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker, and Sheila A. Haghighat. *A domain and Type Enforcement UNIX* prototype*. http://www.usenix.org/publications/library/proceedings/security95/full_papers/badger.pshttp://www.usenix.org/publications/library/proceedings/security95/full_papers/badger.ps.

[DSSSL] James Clark. *Document Style Semantics and Specification Language (DSSSL)*. http://www.jclark.com/dsssl/.

[DSSSLforXML] Didier Martin. *DSSSL for XML: Why not?*. http://www.xml.com/pub/a/2000/05/03/dsssl/. May 200.

[Ferraiolo92] David Ferraiolo and R. Kuhn. *Role-Based Access Control*. Proceedings of the 15th National Computer Security Conference. 1992.

[Ferraiolo95] David Ferraiolo, R. Kuhn, and J.A. Cugini. *Role-Based Access Control (RBAC): Features and Motivations*. Proceedings of the Computer Security Applications Conference. 1995.

[GrsecurityLSM] *Why Grsecurity cannot use LSM*. http://grsecurity.net/lsm.php.

[GrsecurityStudy] Brad Spengler. *Detection, prevention and Containment: A study of Grsecurity*. http://grsecurity.urc.bl.ac.yu/grsecurity-slide.ppt.

[LaPadula1973] Leonard J. LaPadula and D.Eliot Bell. Copyright © 1973 The MITRE Corporation. *Secure Computer Systems: A Mathematical Model*. MITRE Technical Report 2547, Volume II. 31. May 1973.

[LaPadula1990] Leonard J. LaPadula. *Formal Modelling in a Generalised Framework for Access Control. Proceedings of the IEEE Symposium on Security and Privacy,Oakland, CA.*. 1990.

[LaPadula1995] Leonard J. LaPadula. *Set Modelling of a Trusted Computer System*. 1995. Essay, in: Information Security: An Integrated Collection of Essays.

[LoscoccoNSATR2001] Peter Loscocco and Stephen Smalley. *Integrating Flexible Support for Security Policies into the Linux Operating System*. http://www.nsa.gov/selinux/papers/slinux-abs.cfm. February 2001.

[LSM] Stephen Smalley, Timothy Fraser, and Chris Vance. *Linux and Linux Security Module (LSM)*. http://lsm.immunix.org.

[MACdefinition] Copyright © 2002 ITsecurity.com. http://www.itsecurity.com/dictionary/mandac.htm.

[MAPbox] Anurag Acharya and Mandar Raje. *MAPbox: Using Parameterised Behaviour Classes to Confine Untrusted Applications*. http://denali.cs.washington.edu/relwork/papers/mapbox.pdf.

[Macok2003] Martin Mačok. *Master's thesis: Dynamické řízení přístupových práv (Czech language)*. April 2003.

[MedusaImpl] Marek Zelem, Milan Pikula, and Martin Očkajak. *Medusa DS9 implementation*. http://medusa.fornax.sk/.

[MLSdescript] Fred B. Schneider. *Multi-level Security*. http://www.cs.cornell.edu/Courses/cs513/2000sp/L11.html.

[MZelem2001] Marek Zelem. *Integration of various security policies to OS Linux (in Slovak language)*. http://www.terminus.sk/~marek/skola/Diplomovka.ps.

[ZelemPikula2001] Marek Zelem and Milan Pikula. *ZP Security Framework*. http://medusa.fornax.sk/English/medusa-paper.ps.

[Proposed-NIST-RBAC2001] David Ferraiolo, Ravi Sandhu, and Serban Gavrila. *Proposed NIST Standard for Role-Based Access Control*. National Institute of Standards and Technology. http://csrc.nist.gov/rbac/rbacSTD-ACM.pdf.

[RSBACLSM] Amon Ott. *Rule Set Based Access Control (RSBAC) for Linux and Linux Security Module (LSM)*. http://www.rsbac.org/lsm.htm.

[RSBACImpl] Amon Ott. *Rule Set Based Access Control (RSBAC)*. http://www.rsbac.org/.

[SELinuxImpl] *Secure-Enhanced Linux*. http://www.nsa.gov/selinux/.

[SmalleyNAITR2001] Timothy Fraser and Stephen Smalley. *A Security Policy Configuration for the Security-Enhanced Linux*. http://www.nsa.gov/selinux/papers/policy-abs.cfm. February 2001.

[Subterfugue] Mike Coleman. *Subterfugue implementation*. http://subterfugue.org/.

[SubDomain] Crispin Cowan, Steve Beattie, Greg Kroah-Hartman, Calton Pu, Perry Wagle, and Virgil Gligor. *SubDomain Parsimonious Server Security*. http://www.immunix.org/subdomain.pdf.

[Systrace] Niels Provos. *Improving Host Security with System Call Policies*. http://niels.xtdnet.nl/papers/systrace.pdf.

[TCPIPlayers] *Internet protocol suite - Layers in TCP/IP stack*. http://en.wikipedia.org/wiki/TCP/IP.

[TEmodel] Stephen Smalley. *Configuring the SELinux Policy*. http://www.nsa.gov/selinux/papers/policy2/x87.html. January 2003.

[XMLapplication] Ramaswamy Chandramouli. *Application of XML Tools for Enterprise-Wide RBAC Implementation Tasks*. National Institute of Standards and Technology. http://csrc.nist.gov/rbac/ACM_XML_Paper_Final.pdf.

[XML] *Extensible Markup Language (XML)*. http://www.w3.org/XML/.

[XSL] James Clark. *Extensible Style-sheet Language (XSL)*. http://www.w3.org/Style/XSL.

Appendix A. State-flow security model extension (SFSME)

A.1. Weakness of the existing DTE/RBAC implementation

There are two kinds of transitions in the existing DTE implementations (SELinux, DTE for Linux)

- *Object type transition* - the new type of the object is based on a pair of types and optionally a security class. The first type is the domain of creating process and the second type is the type of a related object (e.g. the parent directory for files)[16]
- *Subject type transition* - the new type of the subject is also based on a pair of types and optionally a security class. The first type is the current domain and the second type is the type of the executable. 17

Due to the subject type transition behaviour, the type of the process is changed only in case of the file execution, i.e. the type transition is invoked. Any other event (object manipulation) doesn't invoke the type transition. It means that the process runs in exactly one separated domain with strictly defined security permissions (e.g. access vectors of a DTE matrix). These permissions represent a set of all the permissions the process needs during its execution in a given domain. On the other hand, real processes do not require all permissions in every possible state, but only a limited subset of them. Therefore a security concept which allows to assign the permissions to each state of a process, should be more fine-grained and tighter than the existing stateless security concepts and implementations.

A.2. State-flow definition

We consider full control over the subjects and the objects. A domain state is defined as any event of a subject accessing an object[18]This event can cause a relevant domain type transition or a role transition (in case of RBAC) to occur. Naturally, the new set of permissions will be related to the new domain or role. It also contains transition permissions needed for the next transition to a new domain state. Hence, the existing security policy for describing the process' behaviour for a given domain is split for each domain state into many sets of the various permissions. The set of the security permissions of a given domain does not depend on the domain itself, but also on the state of the domain of the current process.

A.3. State-flow security server

The implementation of the security server has to satisfy the following conditions:

[16]If the process with type *sshd_t* creates or modifies the files in the directory with type *tmp_t*, the result type of the files will be *sshd_tmp_t*.

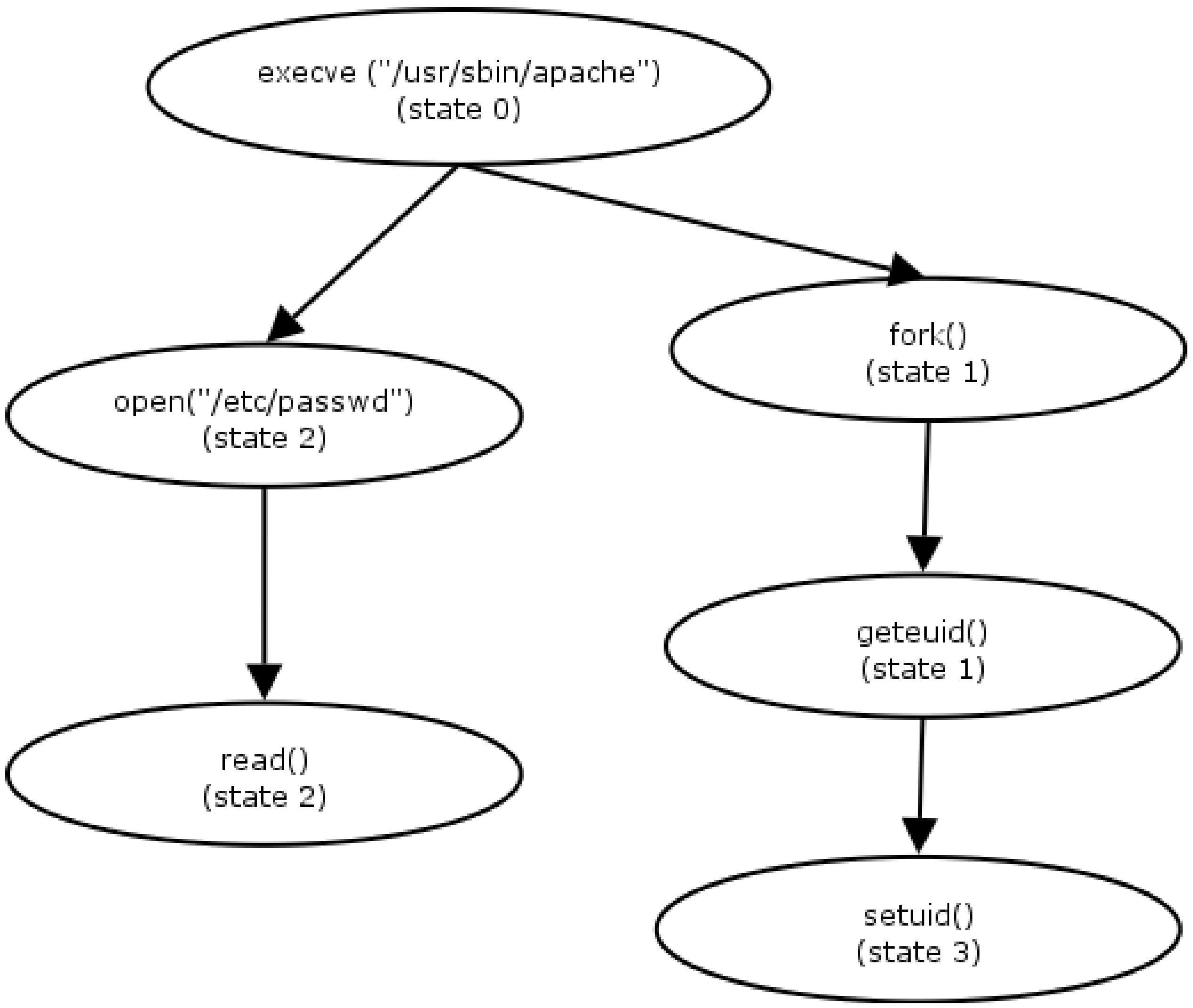
[17]If the process with type *sshd_t* runs executable file with type *shell_exec_t*, the result type of the process will be *shell_exec_t*.

[18]In the case of Linux, a kernel syscall.

- each access of a subject to an object can be a transition event[19]
- each state of a domain can be a full-featured domain with a separate set of permissions[20]

We consider two cases of subject's access to an object. Access which changes the state of a domain (called a transition event) and access which does not change the state of a domain [21].

Figure A.1. State-flow diagram of a process with transitional and non-transitional events



[19]Medusa Constable satisfies this condition. In case of a SELinux, the transition event is *execute*.

[20]A set is represented as an access vector of a DTE matrix in case of SELinux or an access type between virtual spaces in case of Medusa.

[21]in SELinux we consider a syscall into the kernel to be an event. The syscalls `open`, `fstat64`, `ioctl`, `fork`, `setuid`, `exec`, `stat64` etc. can be considered as transition events and the remaining syscalls `read`, `geteuid32`, `getegid32`, `getpid`, `time` can be considered as non-transitional events.

For each state of a domain, a separate set of permissions exists, including the transition permission.

An implementation of the security server has to distinguish between these events and change the state of the domain appropriately.

A.4. State-flow security policy

The creation of the correct security policy for each state of the domain could be a complex problem. It is necessary to create a unique state graph, find the appropriate transition events, know how to detect equivalent states and apply the appropriate aggregation and generalisation of input parameters of the transition events.

In case of Linux, we can consider an automatic monitoring system which can collect the information about the transition events using:

- *syscall* `ptrace` - a non-transparent solution (signal handling of SIGCHLD can be modified during ptrace-ing of a process)
- *syscall hooks into the kernel* - a transparent solution, requires a kernel modification or a kernel module

The monitoring system should be able to distinguish between transition and non-transitions events and therefore choose the appropriate set of permissions for each domain state.

A.4.1. Aggregation of the generated states

Since the new state graph can contain a huge number of generated states (which can be memory-consuming and can cause vastness and complexity), we propose an intuitive aggregation that reduces a lot of redundant states into a lesser number of states. The proposed aggregation rules intuitively follow the reduction of the state graph.

Since each transition event invokes a new domain state, there is a need to detect equivalent states and apply subsequent aggregation on the resulting state graph of the entire domain. The aggregation should follow these rules:

- It is possible to aggregate all neighbouring states whose descendants do not branch. If a state branches, we follow all new states. If the new states are equivalent, we aggregate them together with the original state.
- It is possible to aggregate all branched states that are descendants of the same state and are *mutually similar*. The mutual similarity is determined by a heuristic method (which takes into consideration the allowed range of the input parameters of the transition event). [22]

[22]If opening `/etc/passwd` invokes a transition to one state and opening `/etc/group` invokes a transition to another state, there is a possibility to aggregate these states together as a transition event `open("/etc", ...)`

- The loop detection in the resulting state graph is required in order to reach its simplest representation - we are looking for the longest repetitive subsequence in the state graph. The process flow can be repetitive after a certain amount of time, which implies the repeating of transition events.

The application of the above-mentioned aggregation to the full state graph results into a suitable solution which can represent the base of the state-flow security policy. Due to the fact that the process behaviour can differ (the configuration parameters of the processes can be different which affect their behaviour), the aggregation with an appropriate level of generalisation is needed for correct functionality and flexibility of the processes.

A.5. The main goal of the State-flow security model extension

Considering domain states of the running processes, we can achieve a higher level of access control. The permissions related to exactly one domain in which the process runs all the time are separated into several states of the domain.[23]

A.6. Related projects

Martin Mačok [Macok2003] in his master thesis analysed various ways of dynamical access rights control. He compared properties and advantages of dynamic access rights control to static rights control on existing Unix implementations - Systrace [Systrace], MAPbox [MAPbox], SubDomain [SubDomain] and Subterfugue [Subterfugue]. He implemented his own state automaton as a security module called StateTrick for the Subterfugue and demonstrated it on the Unix *pine* email client. An execution of the email client is separated into the two states. In the first (non-interactive) state an user can compose emails, but can not send them or execute external programs. The second state allows user to create SMTP connection or execute external program (`/usr/sbin/sendmail`). There is also strictly defined a time delay of 30 seconds between the first and the second state. A complex dynamic control can be reached by the combination of multiple Subterfugue tricks (modules).

[23]For example the process `traceroute` will parse user input exactly in one state, in another state it will use the CAP_NET_RAW capability and creates raw sockets, etc. There will be a strictly defined set of permissions for each state, i.e. it will be impossible to use the CAP_NET_RAW capability during the parsing of the user input.