Mandatory Access Control

Håkan Lindqvist

May 18, 2006
Master's Thesis in Computing Science, 20 credits
Supervisor at CS-UmU: Jonny Pettersson
Examiner: Per Lindström

Umeå University
Department of Computing Science
SE-901 87 UMEÅ
SWEDEN

Abstract

This thesis discusses the Mandatory Access Control security model, using both a theoretical and a practical approach. Necessary basic security theory and theory concerning security policies are discussed. A formal policy algebra, based on set theory and logic, is discussed and used to specify a policy which is implemented using the SELinux security framework. The SELinux security framework is briefly discussed and its control model explained.

Contents

List of Figures

Chapter 1

Introduction

Most, if not all, computer systems in use in the world today are insecure in some respect. Research has shown that the security technology currently deployed in the computer industry today is unable to provide a sufficient level of protection for most systems [Petb]. Acoording to [Petb] all systems use a security model that is inherently nearly impossible to secure: discretionary access control, or DAC. In that security model, the owner of an object in the system, such as a file, has full control of whom may access it. This open a wast amount of ways in which the system can be rendered insecure due to abuse, accidents or misconfiguration. It is argued in [Petb] that instead the mandatory access control, or MAC, security model should be used. In that model the right to access objects is left exclusively to the operating system and can not be circumvented by the users of the system. MAC implemented on an operating system level defines an access policy in a system that, if defined correctly, is impossible to circumvent, hence the argument is that it provides a greater level of protection.

With this as a motivation, this thesis will introduce basic definitions of security theory, such when a system may be considered secure, in Section 4. Then it is shown how it is possible to establish when a system in secure in Chapter 6 to 7. Finally an example on how established criteria can be implemented, following the criteria presented in Chapter 4.2.3, is presented in Chapter 9 using an existing technology. Hence, the larger part of the thesis will have a theoretical focus.

Chapter 2

Problem Description

2.1 Problem Statement and Goal

The main purpose of this thesis is to examine the `Mandatory Access Control` security model, from both a theoretical and a practical perspective. To make this possible, the basic theory needed to understand the model will be investigated and briefly discussed.

The theoretical approach should, if possible, discuss an algebra that can be used to express the security relations for any system. Such an algebra should be able to describe access restrictions as well provide an abstraction from real systems.

The theoretical survey should then be used to explore the relatively new SELinux MAC framework from the National Security Agency (NSA), an intelligence agency located in the USA. The framework should be used to implement a well defined security setup, defined with the help of the previously mentioned security algebra. The security configuration displayed using the framework need not be large, merely describe some important features, which based on current work are deemed important to display.

The theoretical and the practical approach should then be compared with respect to expressiveness to see if any of them is apparently more expressive than the other.

After reading the thesis, the reader should have a basic understanding of real world application of security.

Chapter 3

Logic and set theory

In the formal aspects of computer security theory, the most common tool is mathematics. More specifically, set theory is used to a large extent to express properties between entities in systems and their relations. This chapter will provide a very basic introduction to the topics set theory and the closely related field of propositional logic. All material presented is adapted from the book "Discrete and Combinatorial Mathematics" [Ral00] and "Mathematical Logic for Computer Science" [Mor01].

3.1 Propositional logic

Propositional logic, or more correctly propositional calculus, work with expressions with two possible values: `true` or `false`. That is, each expression has only two possible values. How such expressions, and the fundamental parts that form them, will be presented below.

3.1.1 Atomic propositions

To assign any meaning to a logical expression, all parts of it must have a defined meaning. The assignment is done through the basic building blocks called atomic proposition. These are the sentences that can not be divided into any smaller compositions, and therefore have a fixed value of `true` or `false`. How compositions of atomic propositions are made is defined in Section 3.1.3. It describes how to apply rules to build "formulas" using operators presented in the following section

3.1.2 Operators

Any expression in propositional logic can be subjected to an operator since the expression only represents a value. The operators either directly affect the value that the expression evaluates to, or take the value and combine it with another value to form a new expression, which in turn gives a new value.

The operators that were described were unary and binary, respectively. Figure 3.1.2 presents relevant operators, in order of precedence.

The interpretation of an operator is easily represented with a truthtable, in which all possible of expression values that the operator uses and the outcome are presented.

Symbol	Type	Name	Example
$\neg$	Unary	Negate	$\neg\, expr$
$\vee$	Binary	Disjunction	$expr1 \vee expr2$
$\wedge$	Binary	Conjunction	$expr1 \wedge expr2$
$\rightarrow$	Binary	Implication	$expr1 \rightarrow expr2$
$\oplus$	Binary	Exclusive or	$expr1 \oplus expr2$
$\uparrow$	Binary	Nor	$expr1 \uparrow expr2$
$\downarrow$	Binary	Nand	$expr1 \downarrow expr2$
$\leftrightarrow$	Binary	Equivalence	$expr1 \leftrightarrow expr2$

Figure 3.1: Operators in basic propositional logic, shown in order of precedence [Mor01]

Operator	Expression	Resulting value	
$\neg$	True	False	
	False	True	

Operator	Expression one	Expression two	Resulting value
$\wedge$	True	True	True
	True	False	False
	False	True	False
	False	False	False

Operator	Expression one	Expression two	Resulting value
$\vee$	True	True	True
	True	False	True
	False	True	True
	False	False	False

Operator	Expression one	Expression two	Resulting value
$\rightarrow$	True	True	True
	True	False	False
	False	True	True
	False	False	True

Operator	Expression one	Expression two	Resulting value
$\oplus$	True	True	False
	True	False	True
	False	True	True
	False	False	False

Operator	Expression one	Expression two	Resulting value
$\uparrow$	True	True	False
	True	False	True
	False	True	True
	False	False	True

Operator	Expression one	Expression two	Resulting value
$\downarrow$	True	True	False
	True	False	False
	False	True	False
	False	False	True

Operator	Expression one	Expression two	Resulting value
$\leftrightarrow$	True	True	True
	True	False	False
	False	True	False
	False	False	True

Figure 3.2: Interpretation of logical operators [Mor01]

The $\leftrightarrow$ operator also hints of what is defined as logical equivalence; two expressions with the same value are considered to be logically equivalent. That is, one can exchange one for the other in some larger, nested, expression and the value of that expression does not change. The notion of nested, large, expression is the topic of the next section.

3.1.3 Formulas

In the previous section, the term "expression" was used extensively to provide context for the operators described. In propositional logic however, the more common term is "formula."

A formula is defined as a syntactically correctly formatted set of operators and identifiers with respect to the following rules, where ::= is syntactic assignment (i.e. what a word can be replaced with, how to interpret the resulting formula is defined using Figure 3.1.2):

1. fml ::= p, for any symbol p that represents some value
2. $fml ::= \neg fml$
3. $fml ::= fml \vee fml$
4. $fml ::= fml \wedge fml$
5. $fml ::= fml \rightarrow fml$
6. $fml ::= fml \leftrightarrow fml$
7. $fml ::= fml \oplus fml$
8. $fml ::= fml \uparrow fml$
9. $fml ::= fml \downarrow fml$

The reader should note that several of the rules are mutually recursive.

3.1.4 Quantifiers

Quantifiers are simple operators that express a count of symbols that holds some property. The two commonly used quantifiers are "for all", $\forall$, and "exists" (at least one), $\exists$. How these are used is best illustrated with an example:

$$\forall x(q \rightarrow p)$$

The above formula is read as: "for all x for which the expression q implies p is true."

$$\exists x(q \rightarrow p)$$

The above formula is read as: "there exists at least one x for which the expression q implies p is true."

3.2 Set theory

Set theory is a mathematical way of grouping abstract objects, which may be logical representations of real objects, and describing relations between them.

This section will start out with the definition of what a set is, and then continue with some smaller elaboration on how objects inside a set may be ordered, forming structures such as a lattice.

3.2.1 Forming sets

A set is simply a grouping of several objects, using some criteria for the grouping. An intuitive set to use as an example is the set of natural numbers, which can be expressed with the use of logic:

$$N = \{x \,|\, \forall x \, \texttt{such that x is a non-negative integer}\} = \{0, 1, 2, \ldots\}$$

3.2.2 Special properties and operators

Since a set is an abstract way of grouping objects, some special characteristics have been associated with them to ensure proper interpretation. The more fundamental properties and operators are described below.

Union

A set is a collection of unique objects, that is, if you form a new set by combining two existing sets which both have some object x, only one x will be present in the new set. Such an action is called a union of two sets and is expressed here with an example:

$$\{1, 2, 3\} \cup \{1, a, a, b\} = \{1, 2, 3, a, b\}$$

Set equality

Normally a set is unordered, which means that two sets that have their members ordered in a different way, but consists of the same members are equal. An example:

$$\{1, 2, 3\} = \{1, 3, 2\}$$

Subset

Sets are often partially equal. That is, they are partly made up of parts that are present in other sets. Two situations arise: they can either contain objects that all are present in some other set, or they can contain objects that are present in some other set and then some objects which are not present int the other set. These situations are called subset and proper subset respectively.

Two examples will make the distinction clear:

$$\{1, 2, 3\} \subseteq \{1, 2, 3, 4, 5\}$$

The above depicts a proper subset relation.

$$\{1,2,3,a,b,c\} \subset \{1,2,3,4,5\}$$

The above depicts a subset relation.

Difference

Different sets may describe different groups of objects, which implicates that their set of members differ. To express the set of objects that forms the set of difference between two sets, the difference operator is used. The following examples should make the semantics clear:

$$\{1,2,3,a,b,c\} \setminus \{q\} = \{1,2,3,a,b,c\}$$

$$\{1,2,3,a,b,c\} \setminus \{1,2,3,a,b\} = \{c\}$$

$$\{1,2,3,a,b,c\} \setminus \{1,2,3,a,b,c\} = \{\} = \varnothing$$

Where $\varnothing$ is the empty set.

3.2.3 Powerset

A powerset describes all the possible ways to combinate the members of a set. As such, the powerset function maps a set to a collection of all subsets that belong to the original set. An example will clearify how the mapping works:

$$\wp(\{a,b,c\}) = \{\varnothing, \{a\}, \{b\}, \{c\}, \{a,b\}, \{a,c\}, \{b,c\}, \{a,b,c\}\}$$

Note how the set equality makes the number of sets lesser than if the order of the elements in a set would matter.

3.2.4 Lattices

Lattices are a very important type of ordered sets, where an ordered set is a set in which the set's members are internally ordered under some relation. Lattices commonly occurs in traditional security theory, and will be shortly revisited in the chapter on `security policies`.

A lattice is a structure which has both a single upper and lower bound, and all other objects can be ordered using some relation between these two. For example, assume that the upper bound is the set: $\{1,2,3,a,b,c\}$, and the lower bound is the set: $\{a\}$. Figure 3.3 shows a resulting lattice under the relation $\subseteq$.

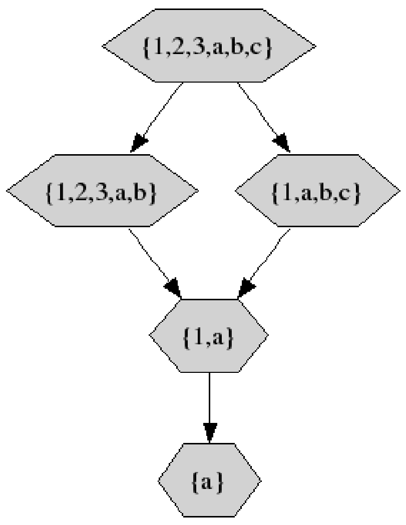

Figure 3.3: A lattice of a few sets under the relation $\subseteq$

Chapter 4

Definitions and foundations of computer security

Mathematical formalism is necessary for the discussion, and therefore there is a need for precise definitions of the entities that are discussed. The more general definitions used will be presented here, with a few examples to make the meaning of each of the definitions clear on an informal level. Unless explicitly stated, all definitions are adopted from [And01a, Bis03].

The discussion in the section on access control matrix constitutes the foundation for the theory of access control, a topic that will be discussed more in later chapters.

4.1 Fundamental components

The widest term used is that of a `system`. It encapsulates the collection of all parts that make up a functional unit of some kind, for example a networked set of computer, their users, administrators, usage regulations and so forth. The definition of a `system` is:

Definition 4.1.1. A `system` is an ordered set made up of, in this order:

1. Products or components
2. Operating system, communications and all things that make up an organisation's infrastructure
3. Applications
4. IT staff
5. Internal users and management
6. Customers external users
7. The surrounding environment

Notice how the definition starts at the core of any organisation's machine setup and works outwards, encapsulating components that are depending on the inner layers. This multilayer view of things is a common way of looking on security, and the importance of each layer will become clear in later chapters. One could say that the level of trust decreases monotonically with each layer. The issue of trust is further elaborated upon in Section 4.2.3.

There are certain entities that issue actions upon other entities. Such an issuer of action is called a `subject`. The definition used is:

Definition 4.1.2. A `subject` is a physical or legal entity in any role. A `subject` s is part of a set S.

Note that this definition encapsulates many types of entities; any organization and persons that are interacting with the system both legally or in an illicit manner. That is, a `subject` is any member of a system from layer 4 and out.

The parts of a system that subjects interact with may either be other `subjects`, or a passive component, called an `object`.

Definition 4.1.3. A `object` is a passive entity in a system, typically acted upon by a `subject`.

The above definitions encapsulates many of the components on a low level in a computer system, such as hardware, files and data structures. A wider definition that encapsulates more entities, from several levels in Definition 4.1.1, is the notion of a `principal` [And01a]:

Definition 4.1.4. A `principal` is an entity that participates in a security system.

This means that a principal includes, but is not limited to, `subjects`, `objects`, processes, a role and any equipment in the system. The difference between a `principal` and an arbitrary entity is that a `principal`, by definition, *interacts* with a security `system`.

4.2 Fundamental properties of a secure system

The basis of computer security issues are made up of three fundamental properties. They describe the accessibility of the system, the correctness of any manipulation of any object on the system and to what extent information considered sensitive is kept secret. The properties are called `availability`, `integrity` and `confidentiality` and they will be described in turn below. The definitions are taken from [And01a, Bis03]. Each section will begin with a discussion, and proceed with the definition of the term.

4.2.1 Availability

One of the most basic aspects of a system is its `availability`. If a subject is unable to utilize the services provided, the service may just as well not exist.

Consider the use of remote surgery using a telesurgical system [Bro]; a surgeon is located at a remote site, possibly in another country, than that of the patient. It is of utmost importance that the responsiveness and accuracy of this system is fast and correct. If the service on either end would cease to respond, catastrophe may very well be the result. In this extreme example of availability, all parts of the resulting system used to perform the operation is of utmost importance to guarantee the wellbeing of the patient. Any interruption in the `availability` of the system's parts will make the `availability` of the entire system to fail.

Definition 4.2.1. Let X be a set of entities and I be a resource. Then I has the property of `availability` with respect to X if all members of X can access I.

Note the use of the word "entities" in the definition. This word gives the definition a bigger encapsulation than if the term `subject` would have been used. This means that any part of the system may judge upon whether some resource I is available or not; it need not be considered a subject to do so. Another, basic but important, thing to notice about this definition is that the `availability` is defined with respect to some well defined set X, but no restriction to its definition is given.

4.2.2 Confidentiality

To keep data and its existence secret is a problem that many organisations put a significant amount of time and money into. To keep data confidential is a major concern to, for example, intelligence agencies and the military, where information is often made available to personnel on a "need to know" basis. Cryptography is an important part in the implementation of confidential systems.

A sensitive example of the use for `confidentiality`, are the medical records that are stored in medical databases. In these databases, information about ones emotional health, inherited genetical diseases, HIV status and more is stored. Most people consider this information to be very private and don't want anybody but perhaps their doctor to know about it. This has good cause as many people have been harassed and fired upon that their medical information has been made public [Sch00].

Definition 4.2.2. Let X be a set of entities and let I be some information. Then I has the property of `confidentiality` with respect to X if no member of X can obtain information about X.

Again, notice the use of the word "entities" and the use of the well defined set X.

4.2.3 Integrity

In most commercial environments, the integrity of information is more important than to protect it from illicit access, although that too is an important issue. Consider for example the importance of integrity in a bank's transaction records or the contents of a gas station's selling record.

There are two main categories of `integrity`–based mechanisms:

– `detective integrity` mechanisms

– `preventive integrity` mechanisms

Detection mechanisms are used to detect any unauthorized modification to information. The mechanism may give a detailed report under which circumstances the information's `integrity` was affected: by whom and what part of the information that was affected, or it may just report that the data has been changed and mark the data as no longer trustworthy.

Prevention mechanisms try to maintain the integrity of any information by blocking any unauthorized attempts to modify it. This also includes the case when a user that has been authorized to modify some information in a certain way tries to alter it in an unauthorized manner.

`Integrity` may be one of the most important issues in today's online world; Schneier [Sch00] means that "integrity is about a datum's relation to itself over time," and gives several examples of why it gets more and more important. He says that today it is very easy to forge an article so that it look genuine, like it came from some well respected magazine. This makes it much more difficult to distinguish false information from valid when obtained from an online source.

Schneier provides a specific example of this, concerning trading with stock: An employee of ParGain Technologies managed to post fake takeover announcements, which were designed to look like they came from the Bloomsberg news service. This had the effect of running the stock up by 30 percent before the truth was unveiled.

Definition 4.2.3. Let X be a set of entities and let I be some information. Then I has the property of `integrity` with respect to X if all members of X trust I.

Yet again, the word "entities" is used with respect to the well defined set X. As was previously noted, the term trust is defined in Section 4.3.1.

4.3 Trust and assurance

In order to implement security in a computer system, it is obvious that there must be one or more system components that can be `trusted` to be correct. One such technique will be discussed in this chapter: `Trusted Computer Base`. After the introduction to the notion of trust, the concept of assurance will be lightly touched upon.

4.3.1 Trust

Trust is probably one of the most important factors in real life: on a daily basis we `trust` in that people will do what they are expected to do in different situations, such as not breaking the law and that everyone will try not to contradict social norms.

For security, it becomes the question [Bis03, Amo94]: `''To what degree can I trust this system being secure?''`

Bishop [Bis03] provides a definition of trust which fully encapsulates the above question:

Definition 4.3.1. An entity is `trustworthy` if there is sufficient credible evidence leading one to believe that the system will meet a set of given requirements. `Trust` is a measure of trustworthiness, relying on the evidence provided.

The definition also captures two key points: that trust is a subjective measure that is dependant on how much evidence of a system's security is provided, and what kind of evidence it is.

So, for a system that has been designed and implemented to be secure, potential users of the system will have to be provided evidence for that any trust they might have for the system's correctness is not misplaced.

This problem is especially true for all companies that sell software; only the trust that a company gets from their customers that they have a functionally correct and secure product will ensure continued investments in software from that company. This brings up the issue of assuring correctness, which is the topic of the following section.

4.3.2 Assurance

As was mentioned in the previous section, there is a need to provide convincing arguments that a product works as stated – that it functions as is claimed and that it is secure. Stated differently, the problem is to provide convincing proof of the system's correctness.

Definition

The process of gathering evidence for the security, and correctness, of a system is called `assurance`. The following definition provides a good starting point for the rest of the discussion [Bis03]:

Definition 4.3.2. `Security assurance`, or simply `assurance`, is confidence that an entity meets its security requirements, based on specific evidence provided by the application of assurance techniques.

The definition does not specify what kind of assurance techniques that should be employed, just that they should. The goal of the assurance techniques though are to eliminate all mistakes that can be done during the development of a system.

Problems and countermeasures

As was mentioned above, the goal of applying assurance is to reduce the occurrence of errors in the system. In that respect, assurance has very much in common with software engineering (see e.g. [Sha, Mar03, And01a]) in that it tries to eliminate as many problem factors as possible. The following list of errors and trouble sources is given in [Bis03] as issues in computer systems:

1. Requirement definition, omissions and mistakes
2. System design flaws
3. Hardware flaws
4. Software flaws, programming and compiler bugs
5. System use and operation errors and inadvertent mistakes
6. Willful system misuse
7. Hardware, communication or other equipment malfunction
8. Environment problem, natural causes and random events
9. Evolution, maintenance, faulty upgrades and decommissions

All of these sources of security issues are well known, and there are different assurance techniques to counter most of them. Natural causes and random events are hard to counter though. The counter measures that are discussed below are adapted from [Bis03].

The first thing to assert correctness for is design. Correctness and completeness testing is an absolute necessity to ensure that the correct security measures are being taken and that they encapsulate the problem space. This kind of assurance deals with problems of type 1, 2 and 6.

Second, the implementation of the designed system must be checked, both hardware and software, which handles the problems listed under 3, 4 and 7. Moreover, this will also handle problems with maintenance, environmental problems and willful misuse (list items 6, 8 and 9) since the assurance should stretch to the deployment of the system.

To counter operational problems, operational assurance can handle the problems under list item 5. Techniques for doing this are, for example, monitoring and audit. These techniques will hopefully find flaws in the system as it is used and is in the maintenance phase, so that those errors can be dealt with [CSR].

Real world example: OpenBSD

As a real world example of how several of the above techniques are being used, the free POSIX–compatible system `OpenBSD`[1] is presented.

`OpenBSD` has a very strong focus on security, and therefore has included a significant amount of well designed countermeasures to well known vulnerabilities such as buffer overflows and rights escalation.

One of the most impressive parts of the `OpenBSD` effort is the thorough code audit the operating system, and its userland tools (i.e. tools used in application space as opposed

[1] Official website: http://www.openbsd.org

to kernel space, in which the operating system's kernel operates), has undergone: the code is constantly checked for bugs and possible errors, which has uncovered a lot of bugs over the years [Ope].

The open nature of `OpenBSD` gives the discussion on design issues a natural solution in the project's mailing lists, flaws in the software are tested for by test releases and actual use, maintenance is a constant factor due to the commitment of the community and the developers.

4.3.3 Certification

Although assurance at different levels while constructing a system is an obvious way of increasing the quality of the end product, different types of assurance techniques are likely to uncover different types of errors and flaws unequally well. To make it possible for someone not involved in the actual work on a system to judge on the process used for verification some common ground between different projects is needed. Certification is such an effort.

A certification is a formalized way of placing trust in a system by using a well defined set of assurance techniques. Different certifications will provide different levels of trust for a `system` based on how the validation is performed.

For example, lower certification levels may require the use of software development methodologies and testing of the correctness of the system's functionality compared to specification documents. On the other end of the scale are formal proof of the systems functionality, in which the entire system is subjected to mathematical proof for the correctness of each line of code.

An industry standard used extensively is the Common Criteria, which has seven levels of trust. The levels ranges from "no trust" to "Formally Verified Design and Tested" [Bis03, IAC]. Very few projects ever reach the higher levels of the criteria.

4.3.4 Trusted Computing Base

Previously in the chapter, the topics of assurance and certification have been discussed. As is easily understood, thorough verification of a piece of software's correctness is time consuming. In commercial projects, cost also becomes an important factor.

For these reasons it is a good idea to minimize the code base that should be responsible for enforcing the security of the system, that is the mechanism that implements the system's security policy.

A definition of the parts of a system that is responsible for implementing the mechanisms is provided [Bis03]:

Definition 4.3.3. A `Trusted Computing Base` (TCB) consists of all protection mechanisms within a computer system – including hardware, firmware and software – that are responsible for enforcing a security policy.

One important conclusion can be reached from this: if the `TCB` is small enough to be properly verified, using techniques such as assurance and certification conformance, and all policy dependant decisions in the system pass through the `TCB`, then the level of trust that can be placed in the system will be directly dependant on the results from the verification of the `TCB`.

The challenge is often to make the size of the `TCB` manageable for any form of verification. It will often include too many parts of an operating system's kernel, and therefore be of great size, to be practically feasible for any more formal type of correctness tests. It is not unusual that the entire memory management subsystem is a part of the `TCB` [Amo94].

4.4 Defining secure systems

One important part of understanding the term "secure" is to understand exactly what that term means in this context. Consider for example the action of opening a file for writing and then writing some data to it. Two questions may come from this seemingly simple action: was the subject allowed to access and modify that subject, and can the system be considered secure after those actions?

Unless explicitly stated, the discussion below is adapted from [Bis03].

4.4.1 Protection states

To be able to analyze a situation in which the question of whether or not a system is secure, the system can be thought of as a state machine in which all components of the system makes up the definition of each state, and changes in any of these components constitutes a transition to another state. Obviously some of the machine's states are undesirable to reach.

To formalize the discussion the notion of a `protection state` is introduced. Let P be the set of `protection states` and Q be a subset of P, where $\forall q \in Q$ are the states that the system is authorized to enter, if the system is to be considered secure.

With the introduction of `protection states`, the problem is now reduced to defining a method that will force the system to only make transitions between states that are in the set Q.

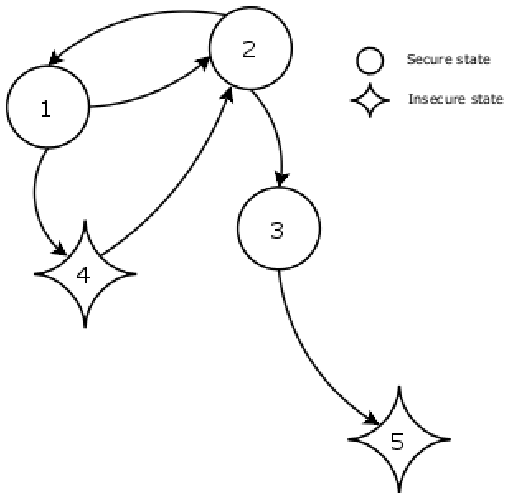

Figure 4.1: A five state situation. The directed arrows depict possible transitions. It is an insecure system since it may reach an insecure state

4.4.2 Definition of a secure system

The follow definition is very general in nature and is quite intuitively tied to the introduction of `protection states` above.

Definition 4.4.1. A `secure system` is a system that starts in an authorized state and cannot enter an unauthorized state.

This entirely encapsulates the example and the discussion in the preceding section; the transitions that would render the state machine describing the system entering an unauthorized state must be avoided if a system is to remain secure.

An example of states and transitions between them is depicted in Figure 4.1. In the figure, states labeled "1", "2" and "3" are defined to be secure, while the states labeled "4" and "5" are defined to be insecure. The system described by these five states, and the possible transitions between them, is not to be considered secure since it is possible for the system to enter an insecure state.

4.4.3 State transition control and the Access control matrix

To solve the problem of which transitions can be allowed, some kind of decision system that takes into account the current context and the requested transition must be constructed. An appropriate model must be able to determine exactly what actions a specific `subject` is allowed to perform on any `object`.

A model that addresses these issues is a very intuitive model called the `Access control matrix model`, which exhaustively describes the rights that each `entity` have over all other `entities` [Bis03].

The model partitions the entities of a system into two categories: the entities that must be protected and those that don't. The entities that are considered relevant to the protection state of the system are placed into either a set of `objects` O or a set of `subjects` S, that is processes and users. This makes it possible to associate all pair (s, o), where $s \in S$ and $o \in O$, with some set of rights $r \subseteq R$, where R is the set of all rights available in the system.

The right r is then captured in an access matrix A such that $a[s, o] = r$, where $a[s, o] \in A$ and by consequence $a[s, o] \subseteq R$.

	file 1	**file 2**	**process 1**	**process 2**
process 1	read	read, write, own	read, write, execute, own	
process 2	read, own	read	read	read, write, execute, own

Figure 4.2: An access control matrix describing the access rights for two processes to two files and each other. Adapted from Bishop [Bis03]

The three sets can be used to describe set of `protection states` for the system using the triple (S, O, A). That is, as long as the identified `subjects`, accesses `objects` accordingly to the rights given by the `access matrix` A, the system will never enter an unauthorized state.

For example, consider the access control matrix depicted in Figure 4.2. It describes exactly which rights "process 1" and "process 2" has with respect to the files "file 1" and “file 2” and to each other. For example, “process 1” has no rights over “process 2,” while the later may read from “process 1.”

4.4.4 Protection states transitions

In the previous section, the access control matrix was introduced. If one assumes that such a matrix is made available, how can the actions necessary to work with `objects` in a realistic way be done in a secure manner? Those actions will necessary include: creation, deletion, modification and rights modification or rights transfer for any object governed by the access matrix.

Bishop [Bis03] describes this as a series of state transitions, in which the protection system changes. More specifically, if a `system` starts in a state $X_0 = (S_0, O_0, A_0)$, and then transitions through a set of states X_1, X_2 ... by the use of operations from an operation set $\{\tau_1, \tau_2, \ldots\}$, then those transitions necessarily change the contents of the access matrix if the access rights of `subjects` are affected, or if `objects` are created or deleted. Obviously there is a need to define the operations in the operation set $\{\tau_1, \tau_2, \ldots\}$ in such way that the access control matrix won't be modified to allow system states that are considered insecure.

These operations may be defined as commands. The definition of the necessary commands has been well defined by previous work by Harrison (among others) [Har76], and those specifications will be used here. Some of the precise notations are adapted from Bishop [Bis03].

The context is such that before the execution of a `primitive command` the system's state can be described by the triple (S, O, A) and afterwards by the corresponding triple (S', O', A'). Special note should be given to the preconditions and postconditions of the aforementioned commands. The precondition presents the condition relative to the system's state for the command to be able to execute, and the postcondition describes the changes done to the access system's state.

1. `Primitive command`: **enter** r **into a[s,o]**
   `Precondition`: The subject and object being considered exist in the access matrix
   `Postcondition`: If the right already exist in that entry, the command will not affect the matrix, else the right r is added

2. `Primitive command`: **delete** r **from a[s,o]**
   `Precondition`: The subject and object being considered exist in the access matrix
   `Postcondition`: If the right does not exist in the access matrix, nothing happens, else it is removed

3. `Primitive command`: **create subject** s
   `Precondition`: The subject must not exist as a subject or an object before the execution of the command
   `Postcondition`: The subject is entered in the access matrix with previous entries having no rights in accessing the new entry. No access entries are added for the new subject either

4. `Primitive command`: **destroy subject** s
   `Precondition`: The subject must exist in the access matrix
   `Postcondition`: The subject is removed from the access matrix, and all entries corresponding to the access rights of others relating to the removed object are also removed

5. `Primitive command`: **create object** o
   `Precondition`: No such object exists
   `Postcondition`: Adds the object to the access matrix. Like the **create subject** command it does not manipulate the access rights present in the matrix

6. `Primitive command`: **destroy object** o
   `Precondition`: The object must exist in the access matrix
   `Postcondition`: The object is removed from the access matrix and all rights in the access matrix relating to the object are also removed

Using these command the access matrix will always be modified in such way that the entries will be left in a consistent state.

However, to make use of these `primitive commands` to represent most real world systems, more preconditions must be evaluated before any of them are to be executed. For example, on most systems only the owner of a file may enter or remove rights of a file. Also, many operations group several `primitive commands` to do their job. A typical example of such an operation is **creating a file**, which might do the following:

1. Does the user have the right to create contents in the directory of interest?

2. Create the file, which is a `object`

3. Set the owner of the file to the creating user

4. Set default rights for everybody else on the newly created `object`

Such grouping must be done for all commands that is done by an operating system [And01b], typically such commands are called system calls.

Considering this new tool, can we be sure that the system is secure? Is the transfer of rights between `subjects` concerning access to other `subjects` and `objects` done in such a manner that no rights are leaked and possibly exploitable by a malicious user? This question is addressed in the next section.

4.5 When is a system secure?

In the previous sections the access matrix, and state transitions that modify it, was introduced. Now the goal is to determine just how such a transition can be made while ensuring that no rights are leaked in an inconsistent manner, enabling the system to enter an insecure state.

This section starts with a light introduction to whether or not an arbitrary `system` may be proved secure. The result of such a proof presents the need for restriction of the transfer of rights in a `system`. Such a restricted model for how rights are appropriately transferred within an access matrix, to ensure that the `system` does not enter an insecure state, is then presented.

4.5.1 Is a system generally provably secure?

A system's security is in the general case, unfortunately, proven to be undecidable. Harrison and Ruzzo [Har76] proved this in 1976 by translating the actions of handling rights in an access control matrix to the state transitions of a Turing machine, and setting the fact that a right has been leaked in the access control matrix as the halting state.

Since it has been shown that a Turing machine can not be proven to reach its halting state, the problem is undecidable since it is non–traceable. (See e.g. [Sud97] for information about Turing machines and the halting problem.)

4.5.2 A model for traceable transfer or rights

Even though it has been proved generally impossible to assert security in system, a more controlled environment than the access control matrix most general form will make it possible to guarantee that a system will remain in a secure state. Sandhu demonstrated such a tool with the Schematic Protection Model (`SPM`) [Rav88], which is proven to be traceable. The non–traceability of the access control matrix is the focus of the problem of that model for general use.

`SPM` is made up of a few simple elements, which when instantiated properly, are able to emulate other protection models. One such model is the `Take grant model` (see [Bis03] for an easy explanation of that model). `SPM` has the following components:

1. A finite set of principal types partitioned into two sets, holding subject's and object's type respectively
2. A finite set of rights, which is partitioned into two sets. One holding the inert rights (the rights that do not alter the system's state) and another holding the control rights
3. A finite collection of links predicates. Used to determine if there is a connection between two subjects, based on their types, that can be used to copy a right
4. A filter function for each link predicate, which will either allow the copying of a right or deny it
5. The operation for creating new subjects and objects
6. A rule that is associated with all operations that create any type of entity, that control which initial rights the created entity will be associated with

Note that SPM as presented only describe a monotonically growing system.

Using this model, all rights are transfered using a well specified system of rules that are unconditionally applied to all operations by the links predicates and the filter functions. Sandhu demonstrates several models in his paper by instantiating the sets in the model's component list appropriately, and gives a convincing argument that many models can be emulated by SPM, and as such be traceable [Rav88]. The fact that SPM is so powerful and proven traceable, makes it clear that models that are either equivalent with or instanciable in SPM will be secure.

4.5.3 Conclusions concerning secure systems

The above discussion made it clear that an unbound system is provably insecure, which can be proven by mapping the primitive operations for modifying an access matrix to a Turing machine.

The introduction of restrictive rules however, made it possible to limit the problem such that it became traceable and thus provably secure. The proof of concept provided was the Schematic Protection Model (SPM).

Since SPM is traceable, it must follow that if a system that has a set of rules for access right transfer, which can be formulated with the SPM, its security is traceable. In other words, if you can express the system's rules in SPM, it is traceable, and hence it can be proven secure; a finite number of operations are needed to ensure that the system never enters an insecure state.

Chapter 5

Access Control

In the discussion about `secure states` in Section 4.4.1, the core issue was transfer of rights. What those rights really describe are the rights any `principal` has to other `principals` in a `system`. The transitions that were described take place as some `subject` access some resource in some way. `Access control` is about regulating which resources that may be accessed and how.

This chapter will begin by defining the two levels that exist for `access control`. The distinction between them is very important since the implications from their specification create two entirely different models for how rights are controlled on a system. After the theoretical foundation has been presented, a few real world examples of the two models are presented, with the security implications and problems that have arisen in those system.

5.1 Definitions of access control specification levels

The specification level of `access control` can be on either of two levels: The specification is up to a `subject` or it is handled by the operating system. The definition of each of these models follow, adapted from [Bis03].

Definition 5.1.1. If an individual `subject` can control the decision of an `access control` mechanism so that it allows or denies access to an `object`, that mechanism is a `discretionary access control` (`DAC`), also called an `identity based access control` (`IBAC`).

Definition 5.1.2. When a system mechanism controls access to an `object` and an individual `subject` cannot alter that access, the control is a `mandatory access control` (`MAC`), occasionally called a `rule-based access control`.

It is important to note that the use of one of these does not prohibit the use of the other. If both `MAC` and `DAC` are used, both `access control` mechanisms must grant a `subject's` right to access an `object`.

5.2 Implications of different types of access control

The implications of `MAC` and `DAC` `access control` mechanisms are more far reaching than one might expect. The most common by far in industry standard operating systems today is DAC in various forms and expressiveness. All variations of UNIX[1] support it, as does Microsoft Windows[2].

The main problem with DAC is that it has too many dependencies for correctness. It depends on the correctness of the policy's specification (cf. Chapter 6), the administrators ability to assure that all objects on the system have rights as specified and the correctness of all tools working with the objects [Petb].

With MAC the main problem is the correct classification of subjects and objects such that correct access rights are enforced. Another issue is the comprehensibility of the policy specification to the MAC mechanisms [Lee95, Ken96, Ray, Peta]. That the implementation specific specification of the policy can be hard to comprehend will become apparent in the chapter on SELinux (cf. Chapter 8).

5.3 Real world examples

The real world is always more complex than what mere theoretical work can express, many factors that pure theoretical settings do not need to worry about must be handled. The provided example of the MAC system is one such example, which is able to meet demands on the enhanced security mechanisms to integrate seamlessly with the existing binary software.

5.3.1 UNIX System V/MLS Access Control

The Unix System V/MLS [Amo94] combines both DAC and MAC seamlessly. The discretionary part resides as is expected in settings associated with the files on the system, while the mandatory part of the access control resides as a part of the operating system's kernel. The fact that the mechanism for MAC resides as an isolated part of the kernel, and is used by system calls, minimizes the intrusion of enforcement code in user space, which is an important aspect.

It is important to note that the completeness of such an approach really depends on the completeness of the hooks inserted into the system. That is, that all parts of the system that change the system's state, with respect to security, pass through the MAC enforcement mechanism. Any relevant execution path that fails to do so will bring the system into an insecure state.

5.3.2 SunOS

Almost all UNIX–like operating systems use DAC as their primary access control mechanism. The SunOS[3] operating system is not an exception. This ensures it, among most existing operating systems today, a place among the implicitly insecure systems, as is argued in [Petb].

[1]UNIX is a registered trademark of The Open Group
[2]Windows is a registered trademark of Microsoft Corporation
[3]SunOS is a trademark of Sun Microsystems, Inc

Chapter 6

Security Policies

Upon a decision of constructing a secure system, or making an existing system secure, there are several issues that must be handled. First one will have to determine what is considered secure, what threats that have to be addressed and formulate the decisions that are reached in a way that is as unambiguous as possible [And01a, Bis03, Mar03]. Preferably this is done at the design phase upon the construction of the to be `system`. The goal in this phase is to identify the secure and insecure states that describe a secure system, as is defined in Definition 4.4.1. The identifications of threats to the system may be done in a formal manner using for example threat trees [Amo94, And01a].

This chapter will start with a presentation of the basic definitions needed to discuss the theory constituting the basis for security policies in a consistent manner. Then some general theory is presented and the chapter is finished with a discussion of security as a process and provide examples of three security policy models.

6.1 Definitions used in security policy theory

To be able to discuss the field of security policies exactly, it is necessary to introduce definitions of the terms used. First, it is important to know the difference between `policy` and `mechanism` [Bis03].

Definition 6.1.1. A `security policy` is a statement that partitions the states of the system into a set of `authorized`, or `secure`, states and a set of `unauthorized`, or `non secure` states.

Again, note the the use of states to define when a system is to be considered secure. Remember that the states referred to are some set of actual settings in a `system` that has been selected to represent its state, such as the `objects` that `subjects` are accessing at a given time or the amount of processing power supplied to an `entity`.

Definition 6.1.2. A `security mechanism` is an entity or procedure that enforces some part of the security policy.

Informally that translates to that a `security policy` is the specification of how a system should maintain a secure state, and what is considered a secure state, while a `security mechanism` is a part of the actual implementation enforcing the specification. This is the same difference as in software development between the design and the implementation stages.

When a security policy is defined, it is usually tailored to the needs of a certain organization. Therefore different policies will have different focal points among the fundamental properties of a secure system (cf. Section 4.1). Many of the more traditional policy models focus on the integrity and/or the confidentiality of a system's information. One such example is given in Section 6.4.1.

6.2 Security as a process

It is all too easy to see a "secure system" as a product. But the truth is that it is more correctly seen as a process. If the bugs discovered in software, and the security advisories that will follow upon these, are taken into account, one easily understands that no system will be perfect from its conception.

In previous chapters, formal security in the terms of states and assignment of access rights have been considered. In a real world problem, those issues often come down to the following steps (adapted from material presented in [Bis03, And01a]):

1. Determine which of the three categories `integrity`, `confidentiality` and `availability` that are relevant for the system

2. Identify threats to the relevant categories, using for example threat trees

3. Formulate a policy that ensures that protects against the threats identified in the previous step

4. Check the policy's completeness (cf. Section 4.2.3)

5. Implement the policy in the system

6. Verify the implementation (cf. Section 4.2.3)

This process may be iterated several times as new threats are identified, the policy is revised or that new rules are set by the management.

It may appear very simple to perform these steps, but it may require quite large amount of work to determine the threats present to a `system` in a real environment, which can be very different depending on who is using it, and then categorizing them in a consistent manner. Moreover, there is always the risk of formulating a policy that is either too restrictive or, on the converse, too non–restrictive.

6.3 Formulation of a security policy on an organizational level

In the previous section, the process of formulating a security policy and implementing was briefly described. This section will attempt to provide some insight to the problems that are introduced when a security policy should be formulated in an organization. As in most problem areas, communication is the most important aspect to consider. The arguments presented herein is in whole based on the organizational theories presented in [Dag98].

6.3.1 Organizational issues when defining a security policy

As has been previously stated, the purpose of a security policy is to define what should and what should not be allowed in an organization. To ensure that this is properly done, the process of defining a security policy should incorporate information from all parts of an organization that will access a `system`. This is important to ensure that the formal restrictions that may be apparent on different levels of the organization are imposed by the formulated policy.

However, most organizations have an hierarchy that defines the responsibilities of a member and a related set of rules for how information should flow within that hierarchy. Such information may be the policy portions that a department wishes to include in a security policy. Hence, to ensure a correct and efficient formulation of a complex policy, the organization must have a good communication infrastructure.

That enough information from each part of an organization is crucial becomes apparent if one considers that the resulting policy should neither impose too much restrictions, since that will affect productivity in a negative way, or be too loose, which will render the policy ineffective.

The collected information should be given to a designated section within the organization to formulate a resulting policy, which then can be implemented. Or course, a security policy can be implemented using a mixture of `security mechanisms` and rules that members of the organization are required to follow.

Another reason to request information from different section of an organization's hierarchy is that it will at least give the members of the organization the impression of that they had a chance to influence what the policy will govern. This will lessen the resistance towards the policy once in place. It will also provide the management, which in the end is responsible for the results of the policy, information on how members of the organization feel about certain issues and what portions of the policy that might need well formulated motivations; rules that seem to be superfluous or counterproductive will be ignored whenever possible.

6.4 Examples of security policy models

The examples presented below range from very classical to more recently developed models. It is important to understand that these models do not make up a policy on their own, a system will need classification of its `principals` to make these work as intended.

6.4.1 The Bell–Lapadula Model

The `Bell--Lapadula Model` (`BLP`) [Bis03] is of traditional military orientation in that it is primarily concerned with maintaining the confidentiality of a system's information.

The central point in the `BLP` is that all `objects` have associated security labels and that all `subjects` have a clearance level. By comparing the clearance with the security labels, access is granted or denied. Since the `system` clears of denies an access, the security model describes a `MAC system`. The clearance and security labels are typically made up of sets on the form:

```
(classification level, {categorization code(s)})
```

An example might be:

`(TOP SECRET, {EU, US})`

for a document classed as "Top secret" and relevant to both Europe and the US.

To understand the idea underlining the model, the `dominate` operation is presented as it is adopted from Bishop [Bis03]:

Definition 6.4.1. The security level (L, C) dominates the level (L', C') iff $L' \leq L$ and $C' \subseteq C$. A `subject` or `object` is said to dominate another `subject` or `object` if its associated security level dominates the other.

By using the dominate relation, properties can be defined such that only information that a `subject` has clearance to read can be accessed, and make downgrading of a piece of information's security classification impossible. The two are:

Definition 6.4.2. Simple Security Condition (No read up)
A `subject` S can only read an `object` O iff S `dominates` O and S has discretionary access to O.

"Discretionary access," means that the access rights associated with the `object` allow that `subject` to access it (cf. Section 5.1 for more information about access control).

Definition 6.4.3. *–Property (No write down)
A `subject` S can write to an `object` O iff O `dominates` S and S has discretionary access to O.

This policy, if correctly enforced, will absolutely maintain the confidentiality of any data in a system. Bishop [Bis03] discusses this model to some length, as well as presenting a few of the more common critiques against it.

6.4.2 Domain Type Enforcement

In a system using `Domain Type Enforcement` (`DTE`) for access control, an access decision to a `subject` or `object` is based upon what domains the involved `subjects` have, and which types the involved `objects` have been assigned [Lee95, Ken96]. Accesses are also controlled by controlling which accesses over domain boundaries `subjects` are allowed to make and which types `subjects` from a certain domain is allowed to access. Expressed in a more casual manner, `subjects` are restricted in their access to a set of `objects` of approved types.

A real world example of such a policy is given in both [Lee95] and [Ken96]. In [Lee95] examples of how the network security is hardened is given, and in [Ken96] a policy that hardens a system's integrity and confidentiality against some well known attacks and some untrusted applications is presented.

That is, `DTE` is used to model which accesses are allowed on what is considered an appropriate type and domain classification of all `principals` in a system. Exactly what policy it enforces depends on which transitions over domain boundaries that are allowed and from which domains `subjects` may access `objects` of different types and with which rights they may do so.

The principal ideas behind this have strong similarities with the Schematic Protection Model presented in Section 4.5.2 in that types of different kinds are associated with all entities in the system and access decisions are based upon those types and a set of rules for interpreting the context upon an access attempt.

6.4.3 Role Based Access Control

An approach that has increased in popularity is to associate access rights to `roles` instead of `subjects`. That is, a `subject` may assume several roles over time, and have access rights changed automatically as transitions are done. It is also possible to make `Role Based Access Control` (`RBAC`)–system [Rav96, Syl00] behave as systems using more traditional access control measures, such as pure `Identity Based Access Control` (`IBAC`, cf. Definition 5.1.1).

In a common model named RBAC96 [Syl00], the following components are used to define roles and their associated access rights:

1. The set of users
   The set of regular roles and the set of administrative roles
   The set of regular permissions and the set of administrative permissions
   The set of sessions
2. A relation for assignment of rights to regular roles and another relation for assignment of rights to administrative roles
3. A relation for assignment of roles to users
4. A relation for assignment of administrative role to users
5. A partially ordered hierarchy of roles (i.e. like a lattice)
6. A partially ordered hierarchy of administrative roles (i.e. like a lattice)
7. A function that maps any session to a single user associated with that session. Another function that maps a session to the set of roles and administrative roles associated with it at a given point in time. The associated set may change over time
8. A number of constraints that will be used to determine which values in the above mentioned sets that are allowed

The idea is to associate a user with a session, and then bind that session to a role, that may change over time, which will be used to associate the user with the rights that are appropriate.

By the introduction of a hierarchy of rules, the concept of inheritance can be used to make more powerful roles inherit the rules of less powerful ones. A typical example is the relation between an engineer and a project's manager. The manager will both need to be able to access data that the engineer has access to as well as data that are of more administrative nature, which means that the rights that should be associated with the manager role encapsulates the rights that are associated with the engineer role. The presentations in [Rav96, Syl00] have many good examples and discussions on the topics briefly mentioned here.

To summarize, `RBAC` models what rights can be given to which `subject` by associating a certain role to it and which access rights that are to be allowed are associated with that role. The policy that it enforces is based upon those assignments.

One of the greater benefits of using this model is that it becomes very easy to change the rights associated with a particular user in the system, since the only action needed is to change the role that user has for the moment. For the same reason it becomes easy to appoint new people to new positions, since all access rights associated with the tasks involved for a special position are associated with that position's role, not with a particular user. This makes revoking rights for the user that previously had that role in the system and assigning them to the newly appointed a simple task, simply change the roles for each of them.

Chapter 7

A common security policy algebra

The chapter on security policies (cf. Chapter 6) discussed the necessity to precisely define what is to be considered secure in a particular system. That chapter also mentions a few example policy models, one of which where `Bell--Lapadula`. The exact formulation of that model makes it clear that it is desirable to have an algebra that would be able to express many different security models as an actual policy.

The purpose of this chapter is to introduce one such algebra which has been developed by Wijesekera and Jajodia [Dum03a]. The algebra builds upon the well understood set theory and propositional logic, as well as recent work in the field of theoretical computer security. The algebra is very general and has the potential to be expressive enough to work with several implementations of policy enforcement mechanisms. In particular, the algebra will be used to specify an example policy in Chapter 9, which will be implemented using the SELinux security framework.

It should be noted that although the algebra presented is developed with a `DAC` system as the expected `access control` model, the power it lends to the user is useful in a `MAC` based system as well (cf. the discussion of `MAC` and `DAC` in Chapter 5).

All content in this chapter is adapted from [Dum03a] unless stated otherwise. It should be noted that the definitions in the section on the algebra's semantics and syntax differ slightly from the original due to apparent mistypings in the original definitions. More information on this issue is presented at the same time as the aforementioned definitions. Some finer points on what appears to be either conventions or mistakes made by the authors of [Dum03a] will be discussed in its own section after the syntax and semantics of the algebra have been presented.

7.1 Motivations

As was mentioned above, it is desirable to use an algebra when specifying a security policy. Using a mathematically sound algebra will ensure that the policy specification can be unambiguously specified, and that its correctness can be verified.

Another, but at least equally strong reason, is that two separate policies that are expressed using the same algebra may more easily be combined in such a way that the resulting policy is compliant with the restriction imposed by the policies that were combined. The presented algebra has explicit support for such actions.

7.2 Algebra basics

The algebra operates on permission sets which it map onto `subjects`. This mapping is performed in an nondeterministic manner, in the same sense as transitions are made in a nondeterministic automaton [Sud97]. Hence, one of several possible mappings is selected from a set, which is due to the fact that which set of permissions that is desirable to use may differ depending on the situation.

Consider for instance a banking clerk. It would not be desirable, from the bank's point of view, to allow a clerk to both write and approve any check. The clerk should only be allowed to either read and approve *or* read and write a check. This kind of mapping is represented by the following functional mapping:

$$(Clerk, \varnothing) \rightarrowtail (Clerk, \{\{check, read\}, \{check, write\}\})$$

The above expression is interpreted such that an initial empty permission set is mapped onto either of the permission sets $\{check, read\}$ and $\{check, write\}$, but not both for a specific check. This kind of mapping is nondeterministic.

7.3 Syntax

The algebra's syntax is very similar to traditional logic in that it uses a set of operators to work on strongly typed entities. This section will begin with definitions of the basic syntax followed by an initial presentation of the operators used to denote actions performed. More on the operators can be found in the semantics section.

7.3.1 Syntax definition

To make the definition of the algebra's syntax more concise, a range of sets are defined. These sets are associated with a number of algebra syntax terminal symbols.

Definition 7.3.1. These sets are associated with the algebra syntax:

– $POL*$ is the set of atomic policies

– $PROP*$ is the set of atomic propositions

– $SETP*$ is the set of atomic (second order) set propositions

– POL is the set of policy terms

– $PROP$ is the set of propositions

– $SETP$ is the set of set propositions

The word "atomic" in the above definition means that the definition of the term at hand is precise and unambiguous. Note the difference between the $*$–suffixed sets with their non–suffixed counterparts.

The next definition associates a number of algebra terminal symbols with the above defined sets.

Definition 7.3.2. The following terminal symbols are associated with the sets defined in Definition 7.3.1.

– p_{atomic} is a symbol taken from $POL*$

– ϕ_{atomic} is a symbol taken from $PROP*$

– Φ_{atomic} is a symbol taken from $SETP*$

The algebra's syntax is defined using `Baccus Naur Format` (BNF) notation [Sud97]. The operators are explained after the definition.

Definition 7.3.3. The algebra's BNF for policies, propositions and set propositions are as follows:

$$\begin{array}{lcl} p & := & p_{atomic} \,|\, p \sqcup p \,|\, p \sqcap p \,|\, p \boxminus p \,|\, p \Lsh p \,|\, (\Phi :: p) \,|\, (p \shortparallel \Phi) \,| \\ & & p \cup p \,|\, p \cap p \,|\, p - p \,|\, \neg p \,|\, (\phi : p) \,|\, (p \shortmid \phi) \\ & & \odot p \,|\, p ; p \,|\, p* \,|\, min(p) \,|\, max(p) \,|\, oCom(p) \,|\, cCom(p) \\ \phi & := & \phi_{atomic} \,|\, \phi \wedge \phi \,|\, \phi \vee \phi \,|\, \neg\phi \\ \Phi & := & \Phi_{atomic} \,|\, \Phi \wedge \Phi \,|\, \Phi \vee \Phi \,|\, \neg\Phi \end{array}$$

The algebra's BNF definition differs slightly from the one presented in [Dum03a]. The authors mistakenly used ϕ instead of Φ for the outer operators. That the second order set propositions Φ is the correct symbol can be deduced from the semantics definitions, which are presented in Section 7.4.

7.3.2 Operators

In the previously defined syntax definition of the algebra, several operators were introduced in the BNF–specification. The following is an introductory presentation of them; exact definition of their functionality can be found in the section on semantics. However, these examples will make it much easier to understand the definitions of the grammar's semantics.

The first distinction between the operators is that there are two kinds: internal and external. The distinction lies in how the operators works with the set of rights associated with a `subject`. An internal operator changes the set of rights while an external operator changes the range of right sets. This means that an internal operator adds or removes rights for a `subject`, while an external operator adds more different sets that may be, nondeterministically, associated with the `subject` or removes such a set.

The running example features a banking clerk and checks. A plus (+) sign means granting of a right, and a minus (−) sign means removal of a right. The only rights that are available are "read" and "write". The only `object` that can have a right tied to it is a "check." The `subject`, the clerk, in all examples is denoted "c."

Note that the definition of the policies in each example, denoted by p and q respectively, have been shortened from the following form:

$$policy\ p \quad : \quad (c, \varnothing) \mapsto (c, \{\{check, +write\}\})$$

to this more concise form to ease reading.

$$p \quad = \quad (c, \{\{check, +write\}\})$$

External disjunction operator: $\sqcup$

The external disjunction operator, $\sqcup$, enlarges the range of permission sets that may be associated with a `subject`. Exactly how this works is best illustrated with an example:

$$\begin{aligned} p_1 &= (c, \{\{(check, +read)\}\}) \\ p_2 &= (c, \{\{(check, +write)\}\}) \\ p_1 \sqcup p_2 &= (c, \{\{(check, +write)\}, \{(check, +read)\}\}) \end{aligned}$$

In the example the resulting policy allows for either read access or write access to a check.

External conjunction operator: $\sqcap$

The external conjunction operator, $\sqcap$, limits the range of permission sets that may be associated with a `subject`. It removes any sets not present in *both* its operands.

$$\begin{aligned} p_1 &= (c, \{\{(check, +read)\}, \{(check, +write)\}\}) \\ p_2 &= (c, \{\{check, +write\}\}) \\ p_1 \sqcap p_2 &= (c, \{\{(check, +write)\}\}) \end{aligned}$$

In the example, the resulting policy only allows for assignment the right to write a check, not to read one.

External difference operator: $\boxminus$

The external difference operator allows any right granted by the first policy after removing any rights defined in the second one. This is very similar to how the familiar difference operator in set theory works (c.f Chapter 3).

$$\begin{aligned} p_1 &= (c, \{\{(check, +read)\}, \{(check, +write)\}\} \\ p_2 &= (c, \{\{check, +write\}\}) \\ p_1 \boxminus p_2 &= (c, \{\{(check, +read)\}\}) \end{aligned}$$

In the example, the write access is removed from the resulting policy since the seconds policy, p_2, states that as a right.

External negation operator: $\Lsh$

This operator changes the right set and replaces it with the relational complement of the current mapping. Again, this is closely tied to the well known field of set theory.

$$\begin{aligned} p &= (c, \{\{(check, -read)\}, \{(check, -write)\}\} \\ \Lsh p &= (c, \{\{(check, +read)\}, \{(check, +write)\}\}) \end{aligned}$$

The resulting policy allows either for read access or write access.

External scoping operator: $::$

The external scoping operator restricts the domain of the original mapping to those permission sets that satisfy the given set proposition[1] representing the scope.

[1]A statement that affirms or denies something and is either true or false

$$\begin{array}{rcl} \Phi & = & \forall y \forall z \left[(y,z) \in X \rightarrow z = +read \right] \\ & & \texttt{where X is a free set variable} \\ p & = & (c, \{\{(check, +read)\}, \{(check, +write)\}\}) \\ \Phi :: p & = & (c, \{\{(check, +read)\}\}) \end{array}$$

The proposition removes the write access right, and hence restricts the available scope. Note that the result is the same as for the external conjunction operator, but in this case the write access is removed by the use of a logical expression which functions like a filter.

External provision operator: $\shortparallel$

The external provision operator, $\shortparallel$, restricts the original mapping to permission sets to those satisfying the set proposition representing the provision.

$$\begin{array}{rcl} \Phi & = & \forall y \forall z \left[((y,z) \in X \rightarrow z = +read) \vee ((y,z) \in X \rightarrow z = +write) \right] \\ & & \texttt{where X is a free set variable} \\ p & = & (c, \{\{(check, +read)\}, \{(check, +write)\}\}) \\ p \shortparallel \Phi & = & (c, \{\{(check, +read)\}, \{(check, +write)\}\}) \end{array}$$

The above example allows the mapping since both read and write actions are approved by the provision.

External sequential operator: ;

The external sequential operator, ;, permits accesses that are allowed as a consequence of applying its second component after the first. This is very similar to how the logical `and` operator works.

$$\begin{array}{rcl} p & = & (c, \{\{(check, +read), (check, +write)\}\}) \\ q & = & (c, \{\{(check, +write)\}\}) \\ p\,;\,q & = & (c, \{\{(check, +write)\}\}) \end{array}$$

External closure operator: $*$

The external closure operator, $*$, allows accesses permitted under repeated application of its constituent policy. This operator is an extension to the external sequential operator.

This operator makes it possible to apply several policies after another an unlimited number of time, it also introduces support for recursive construction on the external level.

Internal disjunction operator: $\cup$

The internal disjunction operator, $\cup$, permits any union of permission sets that are allowed under both its components. That is, the target permission set is altered.

$$\begin{array}{rcl} p_1 & = & (c, \{\{(check, +read)\}\}) \\ p_2 & = & (c, \{\{(check, +write)\}\}) \\ p_1 \cup p_2 & = & (c, \{\{(check, +read), (check, +write)\}\}) \end{array}$$

Internal intersection operator: $\cap$

The internal intersection operator, $\cap$, permits any intersection of permission sets that are allowed under both its components.

$$\begin{array}{lcl} p_1 & = & (c, \{\{(check, +read)\}\}) \\ p_2 & = & (c, \{\{(check + read), (check, -write)\}\}) \\ p_1 \cap p_2 & = & (c, \{\{(check, +read)\}\}) \end{array}$$

Internal difference operator: $-$

The internal difference operator, $-$, permits set differences between its first and second component. Just as the external variation, it is very similar to how the set theory work.

$$\begin{array}{lcl} p_1 & = & (c, \{\{(check + read), (check, -write)\}\}) \\ p_2 & = & (c, \{\{(check, +read)\}\}) \\ p_1 - p_2 & = & (c, \{\{(check, -write)\}\}) \end{array}$$

Internal negation operator: $\neg$

The internal negation operator, $\neg$, changes positive permissions to negative and vice versa.

$$\begin{array}{lcl} p & = & (c, \{\{(check + read), (check, -write)\}\}) \\ \neg p & = & (c, \{\{(check, -read), (check, +write)\}\}) \end{array}$$

Internal scoping operator: :

The internal scoping operator, :, allows only those accesses that meet the scoping restrictions.

$$\begin{array}{lcl} \phi & = & (y, z) \in X \rightarrow z = +write \\ & & \texttt{where X is a free variable} \\ p & = & (c, \{\{(check, +read)\}, \{(check, +write)\}\}) \\ \phi : p & = & (c, \{\{(check, +write)\}\}) \end{array}$$

Since there is a permission tuple in the policy's assignment that provides something else then $+write$, the resulting assignment is non–empty.

Note that the scoping removes rights *before* the mapping, while the provision operator removes them *after*.

Internal invalidate operator: $\odot$

The internal invalidate operator, $\odot$, removes all permissions granted under that policy.

$$\begin{array}{lcl} p & = & (c, \{\{(check + read), (check, -write)\}\}) \\ \odot p & = & (c, \{\{\}\}) \end{array}$$

The resulting permission set becomes the empty set since all permissions are invalidated, that is removed.

Internal provision operator: $|$

The internal provision operator, $|$, allows those permissions that satisfy the provision denoted by a specified proposition.

$$\begin{array}{rcl} \phi & = & (y,z) \in X \rightarrow z = +\mathit{write} \\ & & \texttt{where X is a free set variable} \\ p & = & (c, \{\{(\mathit{check}, +\mathit{read})\}, \{(\mathit{check}, +\mathit{write})\}\}) \\ p|\phi & = & (c, \{\{(\mathit{check}, +\mathit{read})\}\}) \end{array}$$

Note that the provision remove rights *after* the mapping, while the scoping operator removes them *before.*

7.3.3 Conflict resolution

Application of both internal and external operators may remove or add access rights that conflict with each other in a permission set. To handle this situation, either the negative or the positive right is selected in a given situation. The `min` and `max` operators handle this.

To handle unspecified permissions either the `permissions-take-precedence` or `denials-take-precedence` policies of rights management are used. Both of these models are supported by the use of two dedicated operators: `oCom` and `cCom`.

The four operators are used when dealing with over– and underspecified policies.

min operator

The `min` operator selects negative permissions over positive permissions.

$$\begin{array}{rcl} p & = & (c, \{\{(\mathit{check} + \mathit{write}), (\mathit{check}, -\mathit{write})\}\}) \\ \mathit{min}(p) & = & (c, \{\{(\mathit{check}, -\mathit{write})\}\}) \end{array}$$

max operator

The `max` operator selects positive permissions over negative permissions.

$$\begin{array}{rcl} p & = & (c, \{\{(\mathit{check} + \mathit{write}), (\mathit{check}, -\mathit{write})\}\}) \\ \mathit{max}(p) & = & (c, \{\{(\mathit{check}, +\mathit{write})\}\}) \end{array}$$

Permissions–take–precedence, oCom

This operator closes the policy under the open world assumption and grants any right that is not explicitly denied.

$$\begin{array}{rcl} p & = & (c, \{\{(\mathit{check} + \mathit{read})\}\}) \\ \mathit{oCom}(p) & = & (c, \{\{(\mathit{check}, +\mathit{read}), (\mathit{check}, +\mathit{write})\}\}) \end{array}$$

The example adds the right to write a check since that right is not explicitly stated.

Denials–take–precedence, cCom

This operator closes the policy under the closed world assumption and denies any right that is not explicitly denied.

$$\begin{array}{rcl} p & = & (c, \{\{(check + read)\}\}) \\ cCom(p) & = & (c, \{\{(check, +read), (check, -write)\}\}) \end{array}$$

As opposed to the `oCom`–operator, the right to write a check is in this example explicitly denied.

7.4 Semantics

On a semantic level, access control policies allow specified `subjects` to execute actions over given `objects`. That is, an access policy precisely defines how `subjects` may interact with `objects`. That this is the case is trivially apparent when considering the access control matrix introduced in Chapter 4.

The following section begins with a few definitions that are used to more concisely define the semantics later on. The definitions are followed by an introduction of simple atomic policies, which are used to define the semantics of the more general non–atomic policies and the semantics of the operators introduced in the previous sections.

7.4.1 Convenient definitions

First, the basic entities are defined in terms of the algebra, several of which are familiar (cf. Chapter 4).

Definition 7.4.1. `Subjects`, `objects`, signed `actions` and `roles` are the basic building blocks of the semantics. Permission sets and authorization triples using our basic building blocks are defined as follows:

1. `Subjects`: Let $\mathcal{S} = \{s_i : i \in \mathcal{N}\}$ be a set of `subjects`
2. `Objects`: Let $\mathcal{O} = \{o_i : i \in \mathcal{N}\}$ be a set of `objects`
3. `Signed actions`: Let $\mathcal{A} = \{a_i : i \in \mathcal{N}\}$ be a set of action terms. Then $\mathcal{A}^{\pm} = A^{+} \cup A^{-}$, where $A^{+} = \{+a : a \in \mathcal{A}\}$ and $A^{-} = \{-a : a \in \mathcal{A}\}$ is said to be the set of signed action terms
4. `Roles`: Let $\mathcal{R} = \{R_i : i \in \mathcal{N}\}$ be a set of roles
5. `Authorizations`: (s,PermSet) is an authorization if one of the following conditions hold:

   (a) `s` is either a `subject` or a `role` and $PermSet \subseteq \mathcal{O} \times \mathcal{A}^{\pm}$

   (b) `s` is a `subject` and `PermSet` is a `role`
   The notation $\mathcal{AU}(\mathcal{S}, \mathcal{R}, \mathcal{O}, \mathcal{A})$ is used to denote the set of all authorizations over $\mathcal{S}, \mathcal{R}, \mathcal{O}$ and $\mathcal{A}$. When there is no ambiguity about $\mathcal{S}, \mathcal{R}, \mathcal{O}$, and $\mathcal{A}$, the notation $\mathcal{AU}$ is used instead of $\mathcal{AU}(\mathcal{S}, \mathcal{R}, \mathcal{O}, \mathcal{A})$

   (c) `Permission-Prohibition Triples`: Any $(s, o, \pm a)$, where `s` is either a `role` or in $\mathcal{R}$ or a `subject` in $\mathcal{S}$, `o` an `object` in $\mathcal{O}$ and `a` is a signed action term in $\mathcal{A}^{\pm}$, is a `permission-prohibition triple`. The set of all `permission-prohibition triples` is denoted as $\mathcal{T}(\mathcal{S}, \mathcal{R}, \mathcal{O}, \mathcal{A})$, which is shortened to $\mathcal{T}$ when $\mathcal{S}$, $\mathcal{R}$, $\mathcal{O}$ and $\mathcal{A}$ are clear from the context

Note that actions are signed to indicate a permission or a prohibition.

For the interpretation of an atomic policy, the notion of a `state` is defined. A state is used to model a per definition secure assignment of access rights for a specific `subject` to a specific `object`. Note the similarity to the discussion on secure systems in Chapter 4.

Definition 7.4.2. For a given set of `subjects` $\mathcal{S}$, `objects` $\mathcal{O}$, `roles` $\mathcal{R}$, `actions` $\mathcal{A}$, propositions $\mathcal{PROP}$ and set of propositions $\mathcal{SETP}$, a `state` is a pair of mappings $(M_{prop}, M_{setProp})$, where:

$$\begin{aligned} M_{prop} &: \mathcal{PROP} \mapsto \wp(\mathcal{T}) \\ M_{setProp} &: \mathcal{SETP} \mapsto \wp(\wp(\mathcal{T})) \end{aligned}$$

As the definition says, a state is defined as a tuple of mappings. The M_{prop} mapping describes the mapping from propositions to all `permission-prohibition triples` that satisfy the proposition. The $M_{setProp}$ mapping describes the mapping from the set of propositions to the `permission-prohibition triples` set that fulfills the set propositions.

Hence the above mapping, a `state` are determined by the set of propositions satisfied in them. By using propositions in the definition of a state, the provision and scoping operators are assigned meaning. The set of all `states` is called `STATES`.

7.4.2 Atomic policies

An atomic policy is a policy that only includes well defined sets, no operators are involved to change any of the operands. Due to this, atomic policies are used as a base case when defining the semantics for operators in the next section.

The interpretation of an atomic policy is defined below.

Definition 7.4.3. An interpretation of atomic policies $M_{AtPolicy}$ is a mapping from $STATES \times \mathcal{POL}* \times (\mathcal{S} \cup \mathcal{R}) \times \mathcal{P}(\mathcal{O} \times \mathcal{A}^{\pm}) \mapsto STATES \times (\mathcal{S} \cup \mathcal{R}) \times \mathcal{P}(\mathcal{P}(\mathcal{O} \times \mathcal{A}^{\pm}))$ satisfying the condition that for any $(s', PermSet') \in M_{AtPolicy}(St)(p)(s, PermSet)$, $s' = s$.

The definition says that an atomic policy maps an assignment of authorizations to a `subject` or `role` in a certain state to a set of authorizations assigned to the same `subject` or `role` in a possibly different state (cf. the definition of a state above in Definition 7.4.2). The condition $s' = s$ ensures that the same `subject` or `role` is being considered in both states.

It is important to note the following inconsistency in the definition: The function maps a tuple to another tuple. The last line of the definition, though, handles the result of the function appliance as if it was a set. This inconsistency surfaces again in the semantics definition below in Section 7.4.3 and is commented on in Section 7.4.5.

To be able to extend the interpretation of atomic policies to non–atomic policies, the notion of negating permission sets needs to be defined.

Definition 7.4.4. If $\mathcal{PS} \subseteq \mathcal{O} \times \mathcal{A}^{\pm}$ is a permission set, then let $-\mathcal{PS}$ denote $\{(o, -a) : (o, +a) \in \mathcal{PS}\} \cup \{(o, +a) : (o, -a) \in \mathcal{PS}\}$. If $r \in \mathcal{R}$ is a role, then $(o, -a) \in r$ and $(o, +a) \in -r$ `iff` $(o, -a) \in r$.

The definition says that negating a right changes a permission to a prohibition and vice versa.

7.4.3 Non–atomic policies

The semantics of non–atomic policies is defined recursively with the definition of an atomic policy as the base case.

Definition 7.4.5. The interpretation of a non–atomic policy, M_{policy}, is defined by using the previous definition of an atomic policy, $M_{AtPolicy}$.

1. $M_{policy}(St)(p) = M_{AtPolicy}(St)(p)$ for all policies `p` and states `St`
2. $M_{policy}(St)(p \sqcup q)(s, PermSet) =$
   $M_{policy}(St)(p)(s, PermSet) \cup M_{policy}(St)(q)(s, PermSet)$
3. $M_{policy}(St)(p \sqcap q)(s, PermSet) =$
   $M_{policy}(St)(p)(s, PermSet) \cap M_{policy}(St)(q)(s, PermSet)$
4. $M_{policy}(St)(p \boxminus q)(s, PermSet) =$
   $M_{policy}(St)(p)(s, PermSet) \setminus M_{policy}(St)(q)(s, PermSet)$
5. $M_{policy}(St)(\Lsh p)(s, PermSet) =$
   $\{(s, PS) : PS \in \mathcal{PS}\} \setminus M_{policy}(St)(p)(s, PermSet)$
6. $M_{policy}(St)(\Phi :: p)(s, PermSet) = M_{policy}(St)(p)(s, PermSet),$
   $if\ (s, PermSet) \in M_{setProp}(St)(\phi)$ else $\varnothing$
7. $M_{policy}(St)(p \shortparallel \Phi)(s, PermSet) =$
   $M_{prop}(St)(p)(s, PermSet) \cap M_{setProp}(St)(\phi)$
8. $M_{policy}(St)(p; q)(s, PermSet) =$
   $\{(s, PermSet) \in M_{policy}(St')(q)(s, PermSet_2)$:
   for some $(s, PermSet_2) \in M_{policy}(St)(p)(s, PermSet)\}$
9. To define $M_{policy}(St)(p^*)(s, PermSet)$, inductively define $M_{policy}(St)(p^n)$ using the following rules:
   (a) $M_{policy}(St)(p^1) = M_{policy}(St)(p)$
   (b) $M_{policy}(St)(p^{n+1}) = M_{policy}(St)((p; p^n) \cup p^n)$
   (c) $M_{policy}(St)(p^*) = \bigcup_{n \in \omega} M_{policy}(St)(p^n)$
10. $M_{policy}(St)(p \cup q)(s, PermSet) =$
    $\{(s, PermSet_p \cup PermSet_q)$:
    $(s, PermSet_p) \in M_{policy}(St)(p)(s, PermSet)$ and $(s, PermSet_q) \in M_{policy}(St)(q)(s, PermSet)\}$
11. $M_{policy}(St)(p \cap q)(s, PermSet) =$
    $\{(s, PermSet_p \cap PermSet_q)$:
    $(s, PermSet_p) \in M_{policy}(St)(p)(s, PermSet) and (s, PermSet_q) \in M_{policy}(St)(q)(s, PermSet)\}$
12. $M_{policy}(St)(p - q)(s, PermSet) =$
    $\{(s, PermSet_p \setminus PermSet_q)$:
    $(s, PermSet_p) \in M_{policy}(St)(p)(s, PermSet)$ and $(s, PermSet_q) \in M_{policy}(St)(q)(s, PermSet)\}$
13. $M_{policy}(St)(\neg p)(s, PermSet) =$
    $\{(s, -PermSet') : (s, PermSet) \in M_{policy}(St)(p)(s, PermSet')\}$
14. $M_{policy}(St)(\phi : p)(s, PermSet \setminus \{(o, a) : (s, o, a) \notin M_{prop}(St')(\phi)\}) = M$
    if $M_{policy}(St)(p)(S, PermSet) = M$
15. $M_{policy}(St)(p|\phi)(s, PermSet) = M$ provided that
    $M_{policy}(St)(p)(s, PermSet) = \{(s, PermSet) \setminus \{(o, a) : (s, o, a) \notin M_{prop}(St)(\phi)\})\}$,
    where $M_{policy}(St)(p)(s, PermSet) = M$

16. $M_{policy}(St)(max(p))(s, PermSet) =$
$\{(s, PermSet_1) : PermSet_1 = PermSet_2 \setminus \{(o, -a) : (o, +a), (o, -a) \in PermSet_2\}$
for some $(s, PermSet_2) \in M_{policy}(St)(p)(s, PermSet)\}$

17. $M_{policy}(St)(min(p))(s, PermSet) =$
$\{(s, PermSet_1) : PermSet_1 = PermSet_2 \setminus \{(o, +a) : (o, +a), (o, -a) \in PermSet_2\}$
for some $(s, PermSet_2) \in M_{policy}(St)(p)(s, PermSet)\}$

18. $M_{policy}(St)(\odot p)(s, PermSet) = (s, \varnothing)$

19. $M_{policy}(St)(cCom(p))(s, PermSet) =$
$\{(s, PermSet_1) : PermSet_1 = PermSet_2 \cup \{(o, -a) : (o, -a), (o, +a) \notin PermSet_2\}\}$
for some $(s, PermSet_2) \in M_{policy}(St)(p)(s, PermSet)$

20. $M_{policy}(St)(cCom(p))(s, PermSet) =$
$\{(s, PermSet_1) : PermSet_1 = PermSet_2 \cup \{(o, +a) : (o, -a), (o, +a) \notin PermSet_2\}\}$
for some $(s, PermSet_2) \in M_{policy}(St)(p)(s, PermSet)$

The definition's items 6 and 7 have been edited to use a proposition from the correct set. The original definition used ϕ, which is apparently wrong since the mapping used to restrict the permissions that are allowed operate using sets.

Moreover, the definition's items 14 and 15 have been edited from the original one, presented in [Dum03a], to what appears to be the intended semantics definition. The corrected expression is equivalent to that presented in an older paper by the authors which only covered internal operators (cf. [Dum03b]).

7.4.4 Clarification of the semantics definition

The single most complex thing about the algebra is the definition of its semantics. Due to this a discussion of a few points that may or may not be hard to grasp is discussed. If in doubt, read the description of the operators in Section 7.3, in which the algebra's syntax was presented. The indented operation of all the operators is presented there.

For all policies, the state is made up of the mappings defined in 7.4.2. This means, that for each application of the policy transformer of the authentication set, there will be two well defined sets of first and second order propositions, respectively. Due to this, one may explicitly state what propositions the policy should fulfill, which is done in the definition of the provision and scope operators.

The reason why the external operators use set propositions and internal operators uses propositions becomes apparent if the definitions are read carefully: The external operators map the proposition onto an entire permission set, while the internal operators map a proposition onto each element in a permission set.

The closure operator, $*$, is mutually recursive with the definition of the sequential operator, ;, and the internal disjunction operator, $\cup$.

7.4.5 Discussion of the definition's inconsistencies

As mentioned in the definitions in the previous sections, some inconsistencies can be found in the definitions used in the algebra. All definitions are cited from [Dum03a], with a few syntactic correction. Correction of the internal provision operator definition was performed by using the older paper [Dum03b].

The algebra's semantic definition begins with the definition of an atomic policy's interpretation. It states that a tuple of several, mathematical, objects that map onto another tuple, which too consists of several mathematical objects. However, the usage of the result from a transformation made by the policy function is treated as if it was a set elsewhere in both the definition of the policy interpretation and in the semantics definition.

Moreover, one may wonder over the use of the set theoretical conventions, disjunction and difference operators in the definition of the external operators: They are supposed to operate on the permission sets, but do not explicitly state that this is the case, but rather that they should operate on the result of the mapping ultimately performed by an atomic policy specification.

However, this may be a problem that arises from different conventions which may be attributed to field specific conventions. It may for example be assumed that the interpretation simply states how the definition should be read, and that the actual structures operated upon are are the intended set of mappings of access rights to `objects` for `subjects`.

7.5 Examples

To clarify the use of the algebra, several simple examples are provided to illustrate the strength of several of the algebra's operators.

7.5.1 A trivial example

This example shows an atomic policy, that is a simple definition of the rights that are assigned to a `subject` over an `object`.

Assume an `object` o is available in a `system` and that a `subject` s will have an interest in reading, writing but not executing this object. The following policy definition assigns those rights as is appropriate.

$$\mathit{policy}\ p \quad : \quad (s, \varnothing) \mapsto (s, \{\{(o, +\mathit{read}), (o, +\mathit{write}), (o, -\mathit{execute})\}\})$$

This is also called an atomic policy; no operators are involved and it can be precisely interpreted using Definition 7.4.3.

7.5.2 Combining access control specification for a user

For a combination of two policies defining the access rights for a user, or `subject`, it is logical that the range of permissions that may be assigned is increased. This is practical, since the logical way to define access rights is to handle them one at a time.

Assume the `subject` is called s, the `objects` of interest are o_1 and o_2 and that the only permissions that need to be considered are the signed permissions *write*, *read* and *execute*.

$$\begin{aligned}
\mathit{policy}\ p &: (s, \varnothing) \mapsto (s, \{\{(o_1, +\mathit{read}), (o_1, +\mathit{write}), (o_1, -\mathit{execute})\}\}) \\
\mathit{policy}\ q &: (s, \varnothing) \mapsto (s, \{\{(o_2, -\mathit{read}), (o_2, -\mathit{write}), (o_2, +\mathit{execute})\}\}) \\
p \sqcup q &= (s, \{\{(o_1, +\mathit{read}), (o_1, +\mathit{write}), (o_1, -\mathit{execute})\}, \\
&\quad \{(o_2, -\mathit{read}), (o_2, -\mathit{write}), (o_2, +\mathit{execute})\}\})
\end{aligned}$$

7.5.3 Conflict resolution with restriction and closing

If in the previous example, both policies had been concerned with the same `object`, and the internal conjunction operator had been used, there would have been a conflict since the access rights would have conflicted. The following policy definitions depict the situation:

$$\begin{array}{rcl} \textit{policy } p & : & (s, \varnothing) \mapsto (s, \{\{(o, +\textit{read}), (o, +\textit{write}), (o, -\textit{execute})\}\}) \\ \textit{policy } q & : & (s, \varnothing) \mapsto (s, \{\{(o, -\textit{read}), (o, -\textit{write}), (o, +\textit{execute})\}\}) \\ p \cup q & = & (s, \{\{(o, +\textit{read}), (o, +\textit{write}), (o, -\textit{execute}), \\ & & (o, -\textit{read}), (o, -\textit{write}), (o, +\textit{execute})\}\}) \end{array}$$

To resolve this, a conflict resolution operator is applied. There are two of them, one giving restrictions precedence, `min`, and one giving permissions precedence, `max`.

$$\begin{array}{rcl} \textit{min}(p \cup q) & = & (s, \{\{(o, -\textit{read}), (o, -\textit{write}), (o, -\textit{execute})\}\}) \\ \textit{max}(p \cup q) & = & (s, \{\{(o, +\textit{read}), (o, +\textit{write}), (o, +\textit{execute})\}\}) \end{array}$$

If some permissions granted are inappropriate and should be removed, the internal scoping operator could be applied. Say, for example, that the right to write to the `object` is undesirable, since it will be executable, is inappropriate. The following use of the internal scope operator removes it. X is assumed to be a free set variable.

$$\begin{array}{rcl} a = \textit{max}(p \cup q) & = & (s, \{\{(o, +\textit{read}), (o, +\textit{write}), (o, +\textit{execute})\}\}) \\ \phi & = & ((y, z) \in X \rightarrow y \neq -\textit{write}) \vee ((y, z) \in X \rightarrow y \neq +\textit{write}) \\ b = \phi : a & = & (s, \{\{(o, +\textit{read}), (o, +\textit{execute})\}\}) \end{array}$$

However, assuming that the only rights in the system are *write*, *read* and *execute*, the resulting policy b is not closed. Using the completion operators for the closed world assumption, `cCom`, a resulting policy that does not provide the unwanted right to *write* to the `object`.

$$\textit{cCom}(b) \quad = \quad (s, \{\{(o, +\textit{read}), (o, -\textit{write}), (o, +\textit{execute})\}\})$$

Note that if the open world assumption had been used, by applying the *oCom* operator instead of the *cCom* operator, permissions not explicitly denied had been granted, undoing the operation performed by the internal scoping operator.

7.6 Multilevel security

Multilevel security policies (i.e. Bell–Lapadula discussed in Section 6.4.1) require support for comparing labels assigned to `subjects` and `objects`. The algebra presented does not inherently support such operations. The method described in this section will provide a useful method to provide limited support for such usage. First the actual working of the policy is observed, and with these observations as basis a complement to the way the algebra is used is presented to extend that behaviour in a non–intrusive way.

Note that this extension has not been formally tested in any way, its soundness is only argued to be correct.

It is assumed that the `cCom` operator will be used to close the resulting policy.

7.6.1 Observations on the algebra's operations

The first thing one should realize when observing the algebra's way of specifying a policy is a transformation of the permission set and that this permission set exactly defines the access rights to an `object` for some `subject`. This closely resembles an access matrix, which fully specifies all access rights all `subjects` have over all `objects`.

Second, it may be noted that this leaves no room for the usage of levels in the algebra at transformation time, that is when an atomic policy is applied.

7.6.2 Simple multilevel security extension

To make the algebra more useful in the case of multilevel security (MLS), a simple extension of the previously defined algebra is presented.

First, the notion of a `security level` and `classification` is described.

Definition 7.6.1. The set of `classifications` is defined to be $\mathcal{C} = \{c_i : i \in \mathcal{N}\}$, where c_i is the `object's classification`

The set of `security levels` is defined to be $\mathcal{L} = \{l_i : i \in \mathcal{N}\}$.

Second, the following definitions describe the coupling of `objects` to `classification` and `subjects` and `roles` coupling to `clearance`.

Definition 7.6.2. Let all `objects` $o \in \mathcal{O}$ be associated with a `classifications` set $\{c : \forall c \in C\}$ and a security level $l \in L$, which is ordered accordingly to the dominate relation in a lattice and l is the highest `classification` that the `object` may have. Let (o, c, l) be the association of a `classification` and a security level with an object.

Definition 7.6.3. Let all `subjects` $s \in S$ be associated with a classifications set $\{c : \forall c \in C\}$ and a security levels $l \in L$, which are ordered accordingly to the dominate relation in a lattice and l is the highest `security level` the `subject` may have. Let (s, c, l) be the association of a `clearance` with a `subject`.

Let all `roles` $r \in \mathcal{R}$ be associated with a security level and a `security level` in the same manner as `objects`. Let (r, c, l) be the association of a `classification` with a role.

Note that the `subject` have been assigned the highest `clearance` that it may operate under. This effectively makes the policy specification coarse with respect to that it does not support a dynamically changing `clearance`. If it seems strange that an `object` potentially could be assigned a dynamic range of `classifications`, consider a directory, which under some policies might be considered to have the `classification` of the highest classified file which it contains.

Third, a set of propositions that fulfill a MLS criteria that is imposed on the `subjects` and `objects` is defined.

Definition 7.6.4. Let $SETMLSP$ be a set proposition that compares the `classification` and security level of an `object` to the `clearance` and `classification` of a `subject`.

The definition of $SETMLSP$ is now used to define a mapping $M_{SETMLSP}$ that is similar to M_{prop} and $M_{setProp}$.

Definition 7.6.5. Given a MLS set proposition, from the class $SETMLSP$, let the mapping $M_{SETMLSP} : SETMLSP \times PermSet \mapsto \wp(T)$ be the mapping of that set proposition to the set of permission–prohibition triples that fulfills the set proposition, which can be obtained from the supplied permission set.

Finally, the following policy mapping is presented to remove the multilevel security portion from the (s, c, l) and (o, c, l) tuples. This must be done so the existing policy mapping correctly handles the policy specification. The following rule is assumed to be applied before any of the operators or any one of the $M_{AtPolicy}$ or M_{policy} is applied. If the mapping types are carefully studied you will see that the resulting type of the mapping is the one presented in Definition 7.4.3.

Definition 7.6.6. An interpretation of MLS policies M_{MLS} is a mapping $SETMLSP \times STATES \times \mathcal{POL}* \times (\mathcal{C} \times \mathcal{L} \times \mathcal{S} \cup \mathcal{R}) \times \mathcal{P}(\mathcal{C} \times \mathcal{L} \times \mathcal{O} \times \mathcal{A}^{\pm}) \mapsto STATES \times (\mathcal{S} \cup \mathcal{R}) \times \mathcal{P}(\mathcal{P}(\mathcal{O} \times \mathcal{A}^{\pm}))$, which is expressed as:
$M_{MLS}(mls)(St)(p)((s, c_s, l_s), PermSet_{mls}) = M_{policy}(St)(p)(s, PermSet)$,
where $PermSet_{mls}$ is a set consisting of $(c_o, l_c, (o, a))$ tuples and $PermSet = \{(o, a) : (s, o, a) \in M_{SETMLSP}(mls)$

This means that all portions of the MLS extensions will be removed by the operation of M_{MLS} and result in a call to M_{policy}. Note that the policy set used in the definition is the set of atomic policies $\mathcal{POL}*$ (cf. Definition 7.3.1). This restriction was deliberately chosen to avoid the need to redefine the semantics for the algebra's operators in this simple extension.

The definition simply removes all permission sets in $PermSet_{mls}$ before calling the M_{policy} mapping to ensure proper access is allowed, with respect to the proposition that generates the contents of the set $SETMLSP$.

7.6.3 Example usage of the multilevel security extension

The following examples illustrates the usage of the M_{MLS} mapping with a specific second order proposition.

The policy is assumed to govern a `system` consisting of two `objects` $o_1, o_2 \in \mathcal{O}$ with the classification $c \in C$, and two `subjects` $s_1, s_2 \in \mathcal{S}$ with assigned `security levels` $l_1, l_2 \in L$, respectively, and classification $c \in \mathcal{C}$. Level l_1 is higher than level l_2, that is $l_1 > l_2$. The access rights present in the `system` are $\pm read$ and $\pm write$.

The MLS policy of the system states is assumed to enforce that a `subject` must dominate (cf. Section 6.4.1) any `object` that it wishes to access. To enforce such a policy, the following predicate should be used during application of the M_{MLS} mapping:

$$\Phi_{MLS} \quad = \quad \forall(x, y, (z, w))[(x, y, z, w) \in X \mapsto (x \subseteq c_o) \wedge (y \leq l_s)]$$

Where X is a free set variable, l_s is the `security level` of the `subject`, and c_o is the classification of the `object`. All of these are made present during the interpretation defined in Definition 7.6.6.

Now it may be assumed that access right have been assigned to each `subject` for each `object`, in three policies p_{MLS}, q_{MLS} and r_{MLS} while using the MLS specific extensions. The resulting policies are to be combined, which is performed by applying the M_{MLS} mapping to each of them before applying the M_{policy} mapping to them to obtain an atomically specified, combined, policy. The policies are presented below:

$$\begin{array}{lcl}
p_{MLS} & : & ((s_1, c, l_1), ((o_1, c, l_2), \varnothing)) \mapsto ((s_1, c, l_1), (c, l_2, (\{(o_1, +write)\}))) \\
q_{MLS} & : & ((s_2, c, l_2), ((o_1, c, l_1), \varnothing)) \mapsto ((s_2, c, l_2), (c, l_1, (\{(o_1, +read)\}))) \\
r_{MLS} & : & ((s_2, c, l_2), ((o_2, c, l_2), \varnothing)) \mapsto ((s_2, c, l_2), (c, l_2, (\{(o_2, -write)\})))
\end{array}$$

All of these policies are now to be translated into their respective atomic form and combined in an appropriate way, that is per `subject`, which is defined in Definition 7.4.3. Note that the input to the M_{MLS} is a bit shortened since it is apparent from the above definition of the policies p_{MLS}, q_{MLS} and r_{MLS} what the following parameters to the mapping should be.

$$\begin{array}{lllll} p & : & M_{MLS}(\Phi_{MLS})(St)(p_{MLS}) & = & (s_1, \varnothing) & \mapsto & (s_1, \{(o_1, +read)\}) \\ q & : & M_{MLS}(\Phi_{MLS})(St)(q_{MLS}) & = & (s_2, \varnothing) & \mapsto & (s_2, \varnothing) \\ r & : & M_{MLS}(\Phi_{MLS})(St)(r_{MLS}) & = & (s_2, \varnothing) & \mapsto & (s_2, \{(o_2, +read)\}) \end{array}$$

Notice how all rights are removed from the mapping q when mapping translating from q_{MLS}. This is due to that the `subjects` level, l_2 being lesser than that of the `object`, l_1.

The policies can after this translation from the MLS extended version be combined using the previously presented internal and external operators (cf. Sections 7.3, 7.3.2 and 7.4).

7.7 Discussion

The algebra presented in this chapter is very powerful if used correctly. However it is very complex and the definitions used to specify its behaviour are non–trivial to grasp. A good idea would probably be to rework this algebra, and then ensure that all definitions uses the standard form of mathematical operators. Such a reworking is actually rather trivial, but it would be too intrusive on the original presentation given in [Dum03a] to be appropriate for this thesis.

Chapter 8

Security Enhanced Linux

SELinux, one of the most active MAC framework for Linux, is a well functioning technology given away for free by the NSA. The framework is very flexible with a clean separation of policy specification and enforcement mechanisms. This fact shows with the relative ease that it can be configured to enforce vastly different kinds of security policies.

This chapter begins with a short presentation of SELinux' history, continues with a discussion of the implementation's architecture and is concluded with a discussion on policy support and a few tasks that are involved in configuring a policy. In particular, the contents of Section 8.5 is important with its description of the security models that are implemented in the SELinux security framework; that section provides an easily accessible overview of how the enforcements mechanisms work together to provide the protection for the `system`.

8.1 Pre–Linux history

SELinux can trace its origin back to a security platform developed from the Fluke microkernel [And01b], Flask [Ray]. The motivation behind the development of Flask was that existing security platforms lacked at least in one of the areas of access right propagation, fine–grained access control and the revocation of previously granted access rights.

Flask remedied all of these limitations by making sure all parts of the system that must conform to a security policy mediated all system actions with a security server. Moreover, the system had a clean separation of policy specification and implementation. That is, the policy was not explicitly hard coded into the system, but was separately specified. In Flask's case, this was done with a set of configuration files. This explicit separation of implementation and specification of a security policy differ greatly with most of the traditional MAC systems that have been implemented, since those systems tended to have the policy hard coded directly into the policy enforcement mechanisms [Bis03, Amo94].

8.2 SELinux history

Originally, the SELinux implementation was a huge patch to the bare kernel explicitly extending a lot of the data structures it used internally to handle operations concerning files, communications etc.

Although, after a presentation of the framework, a suggestion from Linus Torvalds set off the Linux Security Modules (LSM) project. The project introduced a set of hook functions into the kernel, which can be used to enforce a quite fine grained access control [Steb]. These hook functions are a set of function pointers that are made a part of relevant data structures in the kernel. The functions specified by using these pointers are supposed to be used when making access decisions.

SELinux has since the introduction of LSM in the Linux kernel been ported to the LSM–based model of access right enforcement.

8.3 Architectural overview, Flask and SELinux

Not very surprisingly, the SELinux architecture is very similar to that of the Flask architecture since it is a direct port of the MAC functionality found in the latter system; most of the architectural components have direct functional counterparts in Flask when comparing to the SELinux implementation.

8.3.1 Major components

Flask, being a microkernel system [Ray], has two major parts that are involved in an access decision: the `object manager`, which handles the kernel level details of some entity in the system such as files or processes, and the `security server`, which handles all decision making on accesses.

Linux, not being a microkernel system, but rather a traditional monolithic kernel [And01b], does not have object managers. The counterparts are called kernel subsystems, and those have the corresponding responsibilities of handling kernel space details of entities in the system. The `security server` is located in the LSM compliant module.

The enforcement of policy decisions are made in the kernel subsystems, from which negotiations with the `security server` are done. The negotiation is performed by the calling of the LSM specified security hook functions.

To speed up operations, a cache with previously calculated policy decisions is maintained. In Flask, each object manager had such a cache. Under SELinux the cache is more centralized to the module due to the LSM compliant design.

Policy revocation and reload was a major concern when the Flask security system was implemented [Ray]. Hence Flask supports reloading of security policies and revocations of previously granted rights during system runtime, this feature is also available under SELinux [Petc].

The `security server` is the one entity that makes policy based decisions under SELinux. Hence the `security server` may be perceived as the larger portion of security subsystem's `TCB`. The other, in kernel, `TCB` parts are the LSM hooks that are used to utilize the `security server`. The rest of the `TCB`–like parts of SELinux are the user land tools used to compile the policy configuration and load it into kernel space.

8.3.2 Access decisions

SELinux works *with* the original Linux `DAC` security model in that SELinux first checks if it can allow an access, and then traditional permissions are checked as well. So, SELinux does not replace the original access security model, but rather adds another layer of security.

Since MAC systems associate a label with all objects in a system, and making an access decision compares that label with the authorization level of a subject that attempts to access the object, the SELinux architecture must have some way of representing labels efficiently so that performance does not suffer too bad due to handling of large amounts of in-kernel data. Since all access decisions in SELinux go through the `security server`, it is used to create an efficient mapping between the needed labels and their more efficient representation. This is needed due to the large amount of `objects` handled by the kernel. To be able to handle this rather high number of `objects` present in a `system`, there are two data types used to map an object to the access rights associated with it: a `security context` and a `security identifier` (SID) [Petc, Peta, McC04].

The `security context` is represented as a string and each of these have a corresponding SID, which is passed back to the kernel subsystems to be used for labeling of `objects` created therein. It should be noted that the SID is not exported from kernel space, and is hence not used to mark files in the file system in any persistent manner. How files are marked are discussed below.

To check whether an attempt to access an `object` should be allowed, a kernel subsystem passes the SID of an object to the `security server`. The SID is translated to a `security context`, which is then used by the `security server` to make a policy compliant decision for the access request originally issued to the subsystem.

8.3.3 Files and persistent state

Since the SID to `security context` mapping above is non–persistent, another type of mapping must be used for persistent objects such as files.

The first solution for storing the `security contexts` for files was to store both the Persistent SID (PSID) to inode mapping and the PSID to `security context` mapping on each file system, in its own dedicated file. Of course this was not a good solution with respect to performance since all accesses to any file's `security context` had to access a separate file [Steb, Peta].

When the 2.6 series of the Linux kernel was released, the implementation was changed to store the `security context` in the extended file system attributes that were introduced in that release [Steb]. The attributes are special settings, which are present in each file's vnode, which enables efficient storage of each file's `security attribute` in a, for each file, more localized manner as opposed to storing each `security context` in a separate file. More so, it is less error prone, since there is no single point of failure as in earlier releases; if the mapping file was corrupted, it would have to be rebuilt.

Although the use of extended attributes to store each file's `security attribute` is more a logical way to store it, it also limits which file systems can be used with SELinux. `ext2`, `ext3` `ReiserFS` and `XFS` file systems supports extended attributes, and have been used at some point with SELinux. One of the most used file systems is `ext3`, the default file system on Fedora Core[1]. Other file systems may or may not be SELinux compliant.

[1]Fedora Core is a trademark of RedHat, Inc.

8.4 SELinux LSM implementation

The LSM implementation of the Flask framework is the current implementation of the SELinux framework, and the one currently supported by the community. The parts described here have been mentioned in previous sections, so some of the contents will be familiar.

8.4.1 The security module's internal architecture

The internal structure of the SELinux security module are made up of six parts, some of which diverse from the original Flask implementation in functionality and existence, and some that correlate directly with the Flask counterpart. A short description of each of these components will be given [Steb]:

1. `Security Server`
   This part of the module implements the Flask architecture's `security server`, handling all policy related decisions

2. `Access Vector Cache` (AVC)
   The cache holds previous access decisions made by the `security server`. The cache is used to enhance the performance, since the decision process is a relatively expensive process

3. `Network interface table`
   Parts of the LSM projects suggestions concerning network interfaces were rejected. The parts rejected forces SELinux to internally keep a mapping between network interfaces and `security contexts`

4. `Netlink notification code`
   Uses the Linux NetLink notification system [Sal, Lin] to notify processes of policy changes. These notifications are used by the user space SELinux library, libselinux, to keep the library's internal state consistent with that of the kernel module's

5. `SELinux pseudo file system`
   This pseudo file system exports the `security server` API to processes. This can be compared with for example the `proc` pseudo file system, which exposes process information to user space

6. `Hook functions' implementation`
   These functions are used to initialize the hook functions specified by LSM throughout the kernel's data structures. These functions constitute the entry points for all access decisions to be governed by the SELinux security model in kernel space. Technically, a hook function is a function pointer in C, the language that Linux is written in

As can be seen in the list, much of the components are there to provide support for user space enforcement of a policy. In other word, to give programs running in user space, that not in-kernel, the necessary tools to function smoothly with the policy that is being enforced by the security mechanisms (cf. Section 6.1.2).

8.4.2 Module initialization

Due to the diversity of subsystems that are controlled by SELinux, the module is not fully configured until the policy specification is loaded by the system program `/sbin/init`. The split of the initialization is directly related to internal initializations of subsystems in the kernel [Steb].

The initialization steps are (a more detailed description may be found in [Steb]):

1. Initial initialization. The module sets up its internal state, initializes the AVC and registers itself as the primary LSM security module in the kernel, making the previous primary security module secondary
2. Initializes the netfilter hooks for control of outgoing packets
3. Initializes the internal network interface table, registers a notifier so that the table will be aware of whether a network interface is brought up or down. This is done so that entries can be removed from the table as is needed. Lastly, it registers with the AVC so that the entire table can by flushed upon a policy reload
4. Initializes the NetLink interface to user space for policy load events. As previously mentioned, this is used by the user space SELinux library to keep an internal state consistent with that in the kernel
5. Initializes the SELinux pseudo file system
6. Initializes everything that needs to be initialized on each mounted file system

Anyone familiar with UNIX[2]–like operating systems will see the correlation between the module's initialization steps and the initialization steps performed during a boot of such a `system`.

8.4.3 Module issues

In case the policy used allows for module unloading, it may be possible to unload the SELinux LSM module. If such an action is performed, any action that creates or changes an `object` will result in that `object` being unlabeled, forcing a manual relabeling of such `objects`. The same is true if a machine with SELinux installed is booted with the security framework deactivated. Luckily a tool for automatic relabelling of files is present in the SELinux distribution, namely `setfiles`, which will update the `security contexts` accordingly to the presently active policy when invoked by an administrator.

[2]UNIX is a registered trademark of The Open Group

8.5 Policy support and its representation in SELinux

SELinux is a rather complex `system`. The great flexibility also introduces a steep learning curve for using the `system`.

In this chapter, the basic components of SELinux are introduced, together with an introduction to the steps needed to configure the framework to enforce a locally specified security policy. It is highly recommended that this section is read carefully since it outlines the security models used in the framework and how they work together to provide protection to a `system`.

8.5.1 Overview of the SELinux security model

The `security server` implemented for SELinux has support for a very wide range of policies. This is due to the model that the server uses internally. SELinux uses a combination of:

– Identity Based Access Control (IBAC, cf. Section 5.1.1)

– Role Based Access Control (RBAC, cf. Section 6.4.3)

– Type Enforcement (TE, cf. Section 6.4.2)

The IBAC portion of the security, is usually implemented using the well known process of presenting some proof for being a specific `subject`, with the right to use some `system` identity, during the login process. The user is then associated with an identity internal to the security server, that is, it is independent of the rest of the system.

The use of IBAC ensures that a precise mapping of an identity can be made onto a RBAC policy. The RBAC policy controls which roles that a user may assume and how such a transition may occur. A `subject` will also be assigned a default role upon logging in to the `system`.

The RBAC specification in SELinux differ from that in ordinary RBAC specifications (cf. [Rav96, Syl00]) in that each role specifies which domains it may enter, and leave the permission assignment for each domain to the TE configuration.

The configuration of the TE policy then explicitly grants all allowed accesses by stating which types may interact with each other. SELinux handles domains different from the traditional TE in that domains are treated as types associated with `subjects` and processes while pure types are associated with `objects` (cf. Section 6.4.2 and [Lee95]). This distinction, though, is only enforced by the policy compiler. By handling types this way, SELinux can use a single table for all access right specifications.

The reason that roles are used to control which TE domains that a `subject` is allowed to be entered, is that it makes the configuration more easy to manage since a user may assume several roles; it provides a reasonable level of abstraction.

An example of this configuration is given in Figure 8.1. In that figure a `subject` claims an identity by presenting an identifier for that identity and some kind of proof of that he/she has access as that identity. The identity is then used to check which roles that the subject may assume accordingly to the RBAC configuration. Those roles in turn state which domains the subject may enter, and those domains together with the TE lookup table state which objects the subject may access. This is done, as explained above, by checking the access permissions between the security context of the `subject` and the security context that is associated with the `object` that the `subject` wishes to.

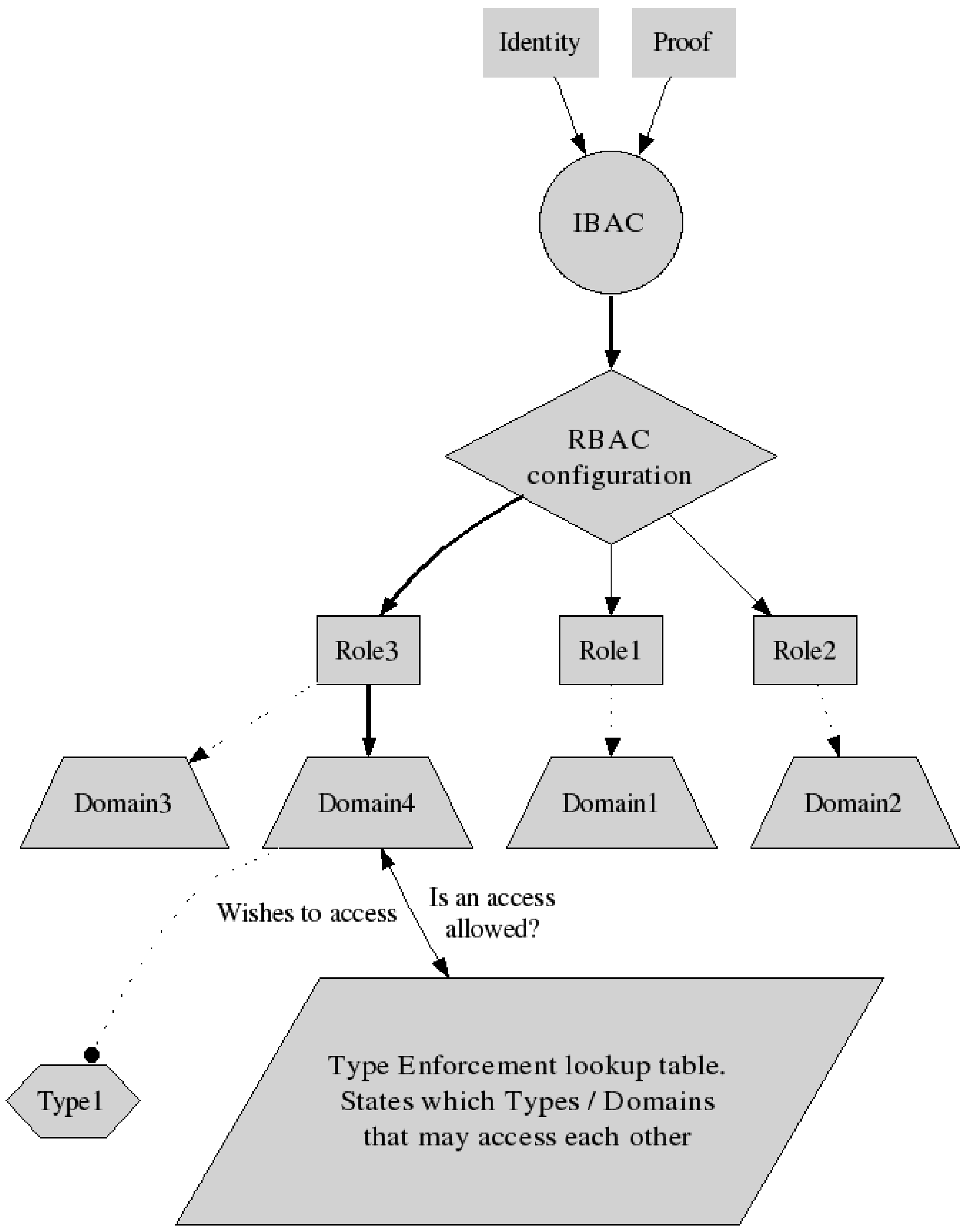

Figure 8.1: A `subject` claims an identity and provides some proof of that he/she has access to that identity. The identity maps into the RBAC configuration which states which roles he/she may assume. The roles that the subject may assume states which domains that may be entered. Assuming that the user entered “Role 1” he/she then attempts to access an object of “Type 1” from “Domain 2”. The access is checked against the TE table with allowed accesses

In the example the subject wished to access an object of type "Type 1" from a domain "Domain 2" while in "Role 1," which presumably is the default role of the `subject`.

Note that just as in standard `RBAC`, only one role may be assumed at a time. If roles should be changed, it must be done explicitly. SELinux provides a command for this called `newrole`.

8.5.2 Policy languages

The interface to the SELinux framework is primarily made up of two parts: Indirectly through the configuration of what security contexts each file on the file system should be labeled, and more directly through the policy language. The policy language is a high level language that is compiled into a binary representation, and then loaded into kernel space using a dedicated program.

It is important to realize that the compiled binary form of the policy is independent of the high level language that is chosen to configure the policy; the factor that controls what language that can be used for configuration is what languages that have compiler support. The current policy language (cf. [McC04] for a rather complete description) is used together with the `m4` macro language [Pro] to provide a rather easy to use scripting interface to the policy configuration. However, recently suggestions have been made as to develop a newer policy language that removes the need to use `m4` macros [pla].

8.5.3 Available SELinux policy configurations

Originally SELinux came with a standard configuration that effectively secured many aspects of the systems that used it. Although the policy had specific example configuration for a set of programs, the policy is generally too restrictive to function unmodified in general. As an answer to this problem, the targeted policy configuration was introduced.

The targeted SELinux configuration secures a number of programs, most notably daemon type programs, such as web and mail servers. The idea is to secure the system one program at a time, making the configuration well fit for the functionality provided by actual systems.

However, a few problems arises with the targeted policy. Currently, the support for multilevel security is not complete. Moreover, it is not as easy to configure as it should be. Especially the later issue motivated the creation of yet another policy formulation, which is built upon the strict and the targeted security policy: the reference policy [Sec]. The reference policy is constructed in a very modular manner, and is well documented. In practice, this is the a reformulation of the strict and targeted policy configuration in a manner that is more easily handled. In the future the goal is to enable the reference policy to handle different portions of the policy configuration as standalone modules, which will make it possible to, for instance, update a specific portion of the policy without the need to recompile the entire policy. The modular organization of the policy makes it more appropriate to use with tools, since each part can be easily edited on its own.

The question that arises is: "Which of the configurations should one select?" The answer varies from case to case. The strict policy will almost certainly require some tuning to function well with an existing system, but will provide much more policy enforced protection than the targeted policy configuration. The targeted policy configuration on the other hand does not provide as complete a protection as the strict one, but does provide an extra layer of security for the locked down programs. The reference policy configuration on the other hand is under active development, and provides most of the protection available under the other configuration, plus additional improvements. The conclusion is that which policy that is selected depends on time, and available expertise for configuration of a policy. There is however very few reasons to not use the reference policy if available.

Much information on actual configurations is available via the `Fedora Core` website [Fed], and the reference policy configuration's website [Sec].

8.5.4 Multilevel security

Aside from the role and type based access control, SELinux has support for the more traditional `Multilevel Security` (MLS) such as Bell–Lapdula as well. This is enforced by adding a `classification range` to each `subject` and `object`, a `clearance range` to each `subject` and a `label range` to each `object`.

The policy grammar can be used to base access decisions on the effective, or current, `clearance` and `label` of a `subject` and the `classification` and `label` of an `object` that is to be accessed. This way, it is possible to enforce that each access is granted in such a way such that the `confidentiality` of any `object` is ensured at all times, as is the case in the Bell–Lapdula policy.

However, the support for MLS in any existing SELinux policies is rather limited due to the fact that this configuration has received the least attention by developers so far. More on available configurations can be found in the following section.

8.5.5 A short overview of SELinux configuration

Configuring SELinux can be a fairly complex task. However, the example configurations that are available provide a good starting ground, as does the documentation. Especially "Configuring the SELinux Policy" by Stephen Smalley [Stea] is a good source of information. The O'Reilly Media has published a good book on SELinux [McC04]. It covers most of the tasks that are involved in configuring SELinux as well as documenting the policy language rather well.

This section will only provide a *very* quick overview of the tasks that (may) have to be performed when changing the SELinux' policy configuration. For detailed information on how SELinux' configuration and implementation works, see [Ray, Peta, Petc, Steb, Stea, Fed]. The mentioned references also contain actual examples on how many of these tasks are performed.

Configuration tasks

The actual configuration of the SELinux policy consists mainly of the following well defined tasks, most of which have been done quite extensively in the example configurations:

1. Adding users. The configuration defines its own users internal to the SELinux' implementation, orthogonal to those of the ordinary Linux system
2. Adding roles. Roles are used to manage which `TE` domains a logged in user may enter
3. Setting up access rights. The inter domain and type access rights must be defined to achieve the access control that is desired

4. Assertions. The use of assertions is to ensure that some accesses are not granted by mistake by the rest of the policy specification

5. Audit level. Accesses that are allowed or denied may or may not be logged. By default only failed accesses are logged to the system log

6. Label the file system. The greatest work is to label the file system. The greatest benefit of using a Linux distribution that provides support for SELinux is that the file system will be configured so that the SELinux tool `setfiles` labels files correctly

8.6 Concluding remarks and other technologies

Although SELinux is a rather mature technology it is lacking in a particular area: the ability to record usage patterns. That is, it is not possibly to record the transitions and accesses needed to perform a certain task, such as typical use of a web browser. Availability of such functionality would no doubt hasten the development of the available SELinux policies since the profiling of executables would be much quicker.

There are tools that may aid the development though, many which are provided by Tresys [tre]. The tools they have produced make it easier to analyze an existing policy configuration as well as develop new configurations. They also have a leading role in the development of the SELinux reference policy configuration.

Support for such profiling can be found in `Systrace` [Sys] and `Novell AppArmor` [Nov], of which the latter is included SuSe Linux 10.0. However, these aforementioned tools then lacks in the area of secrecy since the technologies are more centered around integrity than SELinux, which offers explicit support for `confidentiality` through the support for traditional MLS.

Chapter 9

Reference implementation of a simple policy

The small reference implementation presented in this chapter utilizes the MLS features of SELinux to enforce a strict Bell–Lapadula `security policy`. First, a hypothetic situation is presented. Second, the policy is described in an informal way. Third, the policy is described with algebra presented in Chapter 7. The chapter is then concluded by a presentation of the steps performed to implement the policy using SELinux.

9.1 Problem environment

The problem at hand occurs in a hypothetical organization, from now on called HO, that has a need to enforce a strict confidentiality policy for a number of documents. These documents will however reside on a server that several people that does not have the required clearance to access these documents have access to.

The goal is to ensure that only the members of HO that have proper security clearance will be able to access the documents that are deemed important.

9.2 Informal policy description

The environment described in the preceding section fit very well into a Bell–Lapadula (BLP) security policy since:

1. There is a need for confidentiality
2. There are clear ways of dividing the members of HO into groups made up of those that have the right to access certain documents and those that do not

If some thought is put into the problem, it is also apparent that the strict BLP is an appropriate solution for implementing the layered security; strict BLP disallows read up and write operations are only allowed on the same security level. The restrictions that should be enforced are depicted in Figure 9.1

Note how the information flow in Figure 9.1 allows for moving information to a level with a higher clearance, but no information flow is allowed in the other direction. This

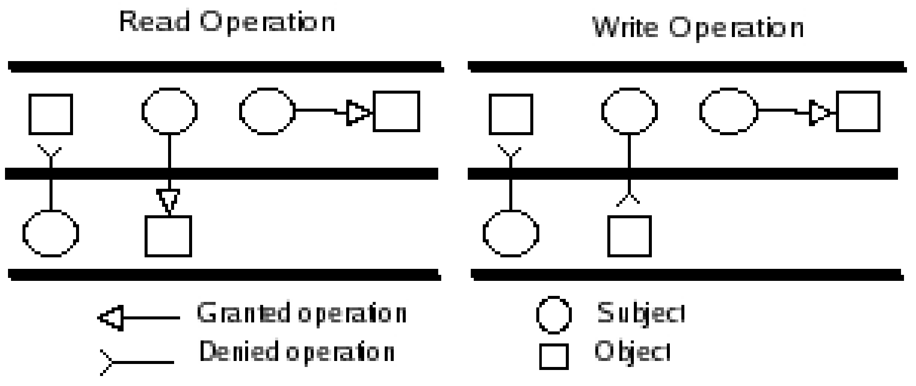

Figure 9.1: Informal description of the strict Bell–Lapadula security policy, only information classed less or equal may be read and write operations can only be performed on `objects` on the same level. Of course, all `objects` that are acted upon must be properly dominated (cf. Chapter 6). The image is adapted from [Amo94]. The two fields between the lines of course represents two distinct security levels

is due to the fact that information may only be written on the same confidentiality level a `subject` is presently on, and read from levels lower or equals the `subject's` own level.

Two assumptions are made : First `subjects` granted access to the `objects` protected by the security policy is assumed not to try to break the policy. One way that will be possible in the implementation, is to copy the files to a remote system that does not enforce the same security policy, and thus downgrade the security. This may of course be avoided by denying the `subjects` that are in a privileged role the access to programs that enables such a course of action. But it is deemed beyond this simple example policy to enforce the policy to such an extent. Second, only directories and files are assumed to need protection, where directories are assumed to hold the property that for the policy specification they can be viewed as files.

Bearing the above in mind, assume that the `subject` that should be able to access the files is called `secret_u`, and that the home directory of that `subject` is `/home/secret/`. Assume also, that the directory `/internal/secret` is the location in which all important documents should be placed. The conclusion is that the aforementioned directories and their contents is the `objects` that should be protected by the policy depicted in Figure 9.1. However, since the programs included in most `systems` are non–compliant with the SELinux environment, in the sense that they do not try to raise or lower their authorization when a user raises or lowers the current level of authorization, the home directory `/home/secret/` should be assumed to have the lowest available classification. The user will log in under the lowest authorization level, enabling the `system` to write to files in the home directory, and then explicitly raise the authorization level and thus drop the rights to write to files in the home directory.

9.3 Formal policy description

Since an informal description often is imprecise, the strict Bell–Lapadula security policy in the preceding section will now be formalized using the algebra from Chapter 7, utilizing the MLS extension introduced in Section 7.6.2. Hence, the rest of this section will introduce each of the elements that are needed accordingly to the definitions in Section 7.6.2. Assumptions that are needed to complete a portion of the policy formulation are introduced in the following sections.

9.3.1 Formulation of the MLS restrictions

The MLS restrictions that governs the accesses between levels, when utilizing the algebra extension introduced in Section 7.6.2, is expressed using a set proposition. This set proposition will then be used to obtain the permissions that are acceptable under these restrictions. Note that the accesses that are of interest must be fully expressed.

$$\begin{aligned}
\Phi_{MLSread} &= \forall(x,y,z,w)[(x,y,(z,w)) \in X \mapsto (x \subseteq c_o) \wedge (y \leq l_s) \wedge (w = +read) \\
\Phi_{MLSwrite} &= \forall(x,y,z,w)[(x,y,(z,w)) \in X \mapsto (x \subseteq c_o) \wedge (y = l_s) \wedge (w = +write) \\
\Phi_{MLS} &= \Phi_{MLSread} \vee \Phi_{MLSwrite}
\end{aligned}$$

The access rights $+read$ and $+write$ are from the set of signed action terms (cf. Definition 7.4.1).

9.3.2 Assignment of MLS authorizations

Since the mapping $M_{SETMLSP}$ uses both a set proposition and a MLS extended permission set to obtain the set of allowed authorizations, it is necessary to specify the needed permissions explicitly at this point.

Assume that the set F is the set of all files on the system and that N is the set of all files that accordingly to the policy should only be readable by a privileged `subject`. The members of the set F, is assumed to have a level of the lowest possible, and are allowed to have any classification that is in the set of all labels. These labels may be shared with those in the set N. The members of set N are assigned the highest level and a label "sSECRET" disjoint from the set of labels that is assigned to the members of F and any classification "c".

Expressed in another way: only the set of files that are categorized to be included in the extra protection that the policy will provide will be labeled and classified in such a way that the access to them will be affected. That is, the rest of the system will more or less be unaffected by the MLS portion of the policy.

This conclusion lead to the following assumption: there is a `system` policy that the MLS portion will coexist with that will govern the accesses to the files that are not explicitly governed by the accesses defined in this example policy. Call this policy p_{system}.

Since SELinux is used, this is actually the case; both the traditional `DAC` model and the used `MAC` SELinux policy will enforce access restriction by the use of `RBAC` and `Type Enforcement` (cf. Chapter 8). The addition of MLS conditions, that due to how the file system is labeled will make certain decisions, will only refine some of the accesses, not all.

SELinux has one other capability: a file that has not an explicit specification of its `security context` inherits it from the parent directory.

The resulting mapping, due to the reasoning presented above and in the previous section, become:

$$\begin{aligned} p_{MLS} \quad = \quad & ((secret_u, sSECRET, c), \varnothing) \mapsto \\ & ((secret_u, sSECRET,), (c, sSECRET, \\ & (/internal/nonpublic, +write) \\ & (/internal/nonpublic, +read)\}) \end{aligned}$$

That this is the case is quite easy to see if one looks carefully at the definitions in Section 7.6.2. Note that the home directory is excluded in the specification; the default policy will handle the user's home directory.

Since it is assumed that there will exist an atomic policy p_{system}, that the policy resulting from the construction in this chapter can coexist with, transforming the above mapping by the use of the previously defined Φ_{MLS}, and the $M_{MLSSETP}$ mapping, will provide the atomic policy that can be used in conjunction with the system policy; p_{system}.

9.3.3 Transforming the MLS extended policy into atomic form

Since the policy p_{MLS} defined in the previous section uses the MLS extensions defined in Section 7.6.2, it must be transformed into an atomic policy using the M_{MLS} mapping.

$$\begin{aligned} p \quad &= \quad M_{MLS}(\Phi_{MLS})(St)(p_{MLS}) \\ &= \quad (secret_u, \varnothing) \mapsto (secret_u, \\ & \qquad (/internal/nonpublic, +write) \\ & \qquad (/internal/nonpublic, +read)\}) \end{aligned}$$

9.3.4 Closing the derived policy

To ensure that proper rights are used under any implementation of the policy described in the previous sections, the policy, p, is closed using the `cCom` operator (cf. Chapter 7). Thus the following operations can be used.

$$\begin{aligned} p_{secure} \quad &= \quad M_{policy}(St)(cCom(p)) \\ \phi \quad &= \quad (x, y) \in X \to x \neq /internal/secret \\ p_{system\,reduced} \quad &= \quad M_{policy}(St)(\phi : p_{system}) \\ p_{system\,secure} \quad &= \quad M_{policy}(St)(cCom(p_{system\,reduced})) \end{aligned}$$

The above use of the internal scope operator will remove any rights concerning the specified directories in proposition ϕ, making access to those impossible after the application of the `cCom` operation.

Using the two policies p_{secure} and $p_{system\,secure}$ will ensure that proper protection is enforced at all times.

9.4 Implementing the policy under SELinux

To implement the policy described in the previous section, the MLS functionality of SELinux will be used. The existing MLS configuration will be activated and extended in such a way that the resulting policy complies with the specified MLS policy. That is, the formal specification from the previous sections is implemented using the security mechanisms provided by SELinux (cf. Chapter 8).

9.4.1 Installation of SELinux support

Installation of SELinux on an actual `system` becomes very distribution dependent. For this reference implementation `Fedora Core 4` (FC4)[1] will be the base `system` used.

FC4 comes with kernel support for SELinux, and essential programs have been patched in the appropriate ways to work with the security framework.

For instruction on how to install FC4, please see the instructions available on the `Fedora Core` web site:

```
http://fedora.redhat.com/docs/fedora-install-guide-en/fc4
```

To add SELinux support after the installation, simply run the following commands, as root, to install the support libraries:

```
# yum install libselinux.i386
# yum install libselinux-devel.i386
```

Furthermore, the reference policy can be downloaded from the from its project website at sourceforge:

```
http://serefpolicy.sourceforge.net/
```

To install the SELinux reference policy configuration, follow the installation instruction available on the “switching page”:

```
http://serefpolicy.sourceforge.net/index.php?page=switch
```

Make sure to download the latest version of the checkpolicy compiler and the libsepol library from the download page, both of which is easily installed using the following commands on FC4:

```
# rpm -U checkpolicy-1.28-4.i386.rpm
# rpm -i libsepol-1.11.7-1.i386.rpm
```

It may also be necessary to ensure that the configuration file for SELinux state that the policy should be run in permissive mode. That is, it should only log what is denied, which is the default logging mode; the security mechanism only log what is being denied, not actually denying any actions. This may be useful to see whether the resulting policy break the usage pattern of any existing software. The configuration file used in the reference implementation, `/etc/selinux/config` is shown below:

```
# This file controls the state of SELinux on the system.
# SELINUX= can take one of these three values:
# enforcing - SELinux security policy is enforced.
# permissive - SELinux prints warnings instead of enforcing.
# disabled - SELinux is fully disabled.
SELINUX=permissive
# SELINUXTYPE= type of policy in use. Possible values are:
# targeted - Only targeted network daemons are protected.
# strict - Full SELinux protection.
SELINUXTYPE=reference
```

[1]Fedora Core is a trademark of RedHat, Inc., visit http://fedora.redhat.com for more information

The mode can be changed to enforcing when the policy has been properly verified.

9.4.2 Setting up the MLS environment

Due to the fact that the MLS support is not fully supported by the current policy configurations, a few special configuration steps must be taken to get a functioning `system`. In this section, all configuration needed to enable the simple MLS policy previously described will be presented in a step–by–step manner. It should be noted that the policy setup is made simpler by the fact that the access specification concerning the MLS portion of the policy specification enforces BLP.

1. Change to the privileged user: `su -`
   The modifications that will follow must be made by a privileged user. On `systems` other than FC4 it may be necessarily to use a `newrole -r system_r` command to assume the `system_r` after changing into the root account. This is due to the fact that the `su` command need not change the role on all platforms. To see if this is the case, simply issue the `id -Z` command, if the role is something other than `system_r`, change roles as was previously described

2. Adding test users.
   The testing of the policy configuration will need three test users: two authorized to access restricted files and one that is not. Hence, these users will have to be added to the `system` before any further configuration is done

   (a) Adding a group for the secret users: `groupadd secret`
   The users that are authorized to access restricted files should be able to utilize the traditional `DAC` security model present in a Linux `system`. Therefore, a group that all such users will be members of is added

   (b) Adding the test users.

   ```
   adduser -G secret secret && passwd secret
   adduser -G secret hidden && passwd hidden
   adduser normal           && passwd normal
   ```

3. `cd /etc/selinux/refpolicy/src/policy`
   The directory in which the source of the reference SELinux policy is located under FC4. If another distribution is used, it may reside elsewhere if not installed directly from source (cf. previous section)

4. Changing build options in `build.conf`
   The file `build.conf` located in the source directory controls a few settings that will affect how the policy is compiled when translating from text to a binary policy that can be handled in kernel space. The following lines are changed:

   (a) `TYPE=targeted-mls`
   The reference policy has, as was mentioned in Section 8.5.3, includes support for both the targeted and the strict policy. Setting the type to `targeted-mls` defines certain macros needed during compilation to use the MLS support currently implemented in the policy configuration as well as activating the security configuration for a large number of programs, most notably a large number of daemon programs (i.e. servers such as web servers)

(b) `DISTRO=redhat`
Fedora Core is sponsored by RedHat, and largely resembles their commercial alternatives, hence the distribution setting is `redhat`. The `DISTRO` option is used to include support for several different distribution, which may have different layout in the file system hierarchy

(c) `DIRECT_INITRC=y`
This option allows a system administrator to explicitly run scripts used during boot. It is activated for convenience

(d) `MONLITHIC=y`
In the future the SELinux policy framework will include the ability to portion the policy up in modules. The modules will, just as real kernel drivers, have the ability to be loaded and reloaded independently of each other [tre]. This policy, however, is of the traditional monolithic type

5. Adding user configuration in `policy/users`
The `policy/users` file contains configuration for the security contexts that `subjects` in the `system` should have. If a user does not have an explicit entry in this file, a generic context will automatically be assigned by the policy during runtime. The following changes are performed to ensure policy compliance:

(a) Restricting the generic user context.

```
ifdef(‘targeted_policy’,‘
gen_user(user_u, user_r sysadm_r system_r, s0, s0 - s14:c0.c255, c0.c255)
’,‘
gen_user(user_u, user_r, s0, s0 - s14:c0.c255, c0.c255)
’)
```

The `gen_user` portion is a macro which is evaluated during compilation. The important part of it is the `s0 - s14` portion, which sets the security levels that are reachable by this user. The lone `s0` part specifies that the user should login at the least privileged level

(b) Adding two test users to the policy, authorizing them to access files that are to be considered restricted.

```
gen_user(secret, user_r system_r, s0, s0 - s15:c0.c255, c0.c255)
gen_user(hidden, user_r system_r, s0, s0 - s15:c0.c255, c0.c255)
```

The difference between these entries and the one for the generic user, is that the user name (i.e. `secret` and `hidden`) are identical to two actual users on the `system` (cf. earlier steps of this configuration) and that the highest security level that they may enter is `s15`, a level which the generic user can not enter

6. Adding file context configuration for the restricted portions of the file system in `policy/modules/kernel/files.fc`.

```
/internal               -d      gen_context(system_u:object_r:default_t,s0)
/internal/secret        -d      gen_context(system_u:object_r:default_t,s15)
/internal/secret(/.*)?          gen_context(system_u:object_r:default_t,s15)
```

The user, `system_u` is a generic user used widely in the policy configuration. The interesting part of the specification is the last part of the call to the macro `gen_context`, the one specifying what security level a `subject` must have to access a file or directory. The first two specify directories and the last on any file under the `/internal/secret/` directory. Hence, any user will be able to access the contents of the `/internal` directory, while the contents of the subdirectory `secret/` will be restricted to users of a high enough security level

7. Disabling of non–MLS compliant programs, edit `policy/modules.conf`
   Due to problems caused by the configuration specific to two programs when installing the compiled policy, those two are disabled. If not, file contexts that are generated for files specific to these programs, simply does not work with the MLS enabled policy, thus causing the auto relabeling of the file system to fail. The programs in question are `screen` and `irc`.

```
# Layer: apps
# Module: irc
#
# IRC client policy
#
irc = off

# Layer: apps
# Module: screen
#
# GNU terminal multiplexer
#
screen = off
```

8. Compiling the policy configuration and installing all necessary configuration files.
   The configuration must be compiled, and file context specifications files generated, and installed under the `/etc/selinux/refpolicy` directory.

```
make clean
make install
```

9. Fixing resulting errors in
   `/etc/selinux/refpolicy/contexts/files/file_contexts.homedirs`
   Due to reasons unknown to the author, an erroneous file context entry is present in the aforementioned file. Hence, this file is edited by hand to adjust the setting to one more appropriate, since the cause for the error could not be found.

```
#/home          -d      system_u:object_r:home_root_t:s15:c0.c255
 /home          -d      user_u:object_r:user_home_dir_t:s0
```

   As can be seen in the specification, the home directory `/home`, would have been configured to be owned by the user root, which obviously is wrong. Therefore, the entry is commented out and replaced by the second line

10. Activate automatic relabeling of the file system: `touch /.autorelabel`
    Creating the file `/.autorelabel` will cause the file system to be automatically relabeled with the correct contexts specified under the `/etc/selinux/refpolicy/contexts` directory

11. Reboot the `system`: `reboot`
    Rebooting the `system` will make sure that the policy can be properly loaded in case a non–MLS policy was previously loaded, and will make sure that all files are properly labeled to work with the policy configuration

One thing to note about the reference policy configuration is that it requires rather new versions of libraries and tools, as was described in Section 9.4.1, which in turn requires a fairly new version of the Linux kernel. If a error on the following form is printed by any tool, or can be found in the log file `/var/log/message`, the kernel should be updated.

```
Mar  2 19:55:13 hostname kernel: security:  policydb version 20 does not
match my version range 15-19
```

The `policydb` portion of the log message refers to a part of the implementation of SELinux security server. If using Fedora Core, updating the kernel is easily done with the command “`yum update kernel`”.

9.4.3 Testing the reference implementation

The previous sections has described the theoretical specification of the strict Bell–Lapdula policy using the algebra, presented in Chapter 7, using the techniques presented in Section 7.6.2, and the actual implementation of it under SELinux using the reference policy configuration. This section will test this configuration to determine whether the implementation behaves in an acceptable manner, allowing only authorized `subjects` to access the files in the directory `/internal/secret/`.

Due to the fact that the testing was performed while using the `X Window System` (X), the policy enforcement had to be turned off during certain operations since the version of X that was used is not compliant with the MLS portion of the policy specification. Hence, any changes that had to be associated with the virtual terminal in use, such as the current security level, could not be updated if the policy was active.

The privileged user secret

The first user to be tested, is the privileged user `secret` added in the configuration steps presented in Section 9.4.2. To indicate whether enforcement of the policy is activated or not, one of the following lines appear in the command listings that follows:

– `# enforce is on`
  This line indicates that root has run the command `setenforce 1`, which activates the enforcement of the currently loaded policy

– `# enforce is off`
  This line indicates that root has run the command `setenforce 0`, which deactivates the enforcement of the currently loaded policy

The user `secret` checks the current identity and security context,and creates a file.

```
# enforce is on
$ whoami
secret
$ id -Z
secret:system_r:unconfined_t:s0-s15:c0.c255
$ pwd
/home/secret
$ echo "A line" > afile
$ ls -lZ afile
-rw-rw-r--  secret   secret   secret:object_r:user_home_t:s0   afile
```

As can be seen from the output, the security level needed to read the file is `s0`, as can be seen in the last portion of the file information printed directly before the filename, “afile”.

Enforcement is turned off, and the working directory is changed to the repository directory `/internal`.

```
# enforce is off
$ cd /internal
$ ls -lZd .
drwxr-xr-x  root     root     system_u:object_r:default_t:s0   .
$ ls -lZ
drwxrwx---  root     secret   system_u:object_r:default_t:s15  secret
```

As is shown in the output from the command, the directory `/internal/` can be read by any user, while the subdirectory `secret/` only can be accessed by a user with a security level of, at least, `s15`. Hence, the enforcement is activated and the last command is ran again.

```
# enforce is on
$ ls -lZ
 ?---------  ?        ?                                        secret
$ cd secret
bash: cd: secret/: Access denied
```

The information about the subdirectory `secret` is no longer available. This is due to the fact that the information which implies the existence of the subdirectory is stored in the parent directory `internal`, while the subdirectory specific information is stored in a separate part of the file system representing the subdirectory. Hence, the user `secret` is able to use the `ls`–command to deduce that there is a directory `secret`, but can get no information from the file system about it since the security level that is tied to the user is too low to authorize access to that information.

The enforcement of the policy is deactivated, and the security level raised.

```
# enforce is off
$ newrole -l s15
Authenticating secret.
Password: <password>
$ id -Z
secret:system_r:unconfined_t:s15-s15:c0.c255
```

The raising of the security level was the only operation that needed the enforcement deactivated, so enforcement is turned on, and a few operations is performed upon the subdirectory `secret`: the user, `secret`, changes the working directory into it and creates two files and a new subdirectory inside it.

```
# enforce is on
# (create two files)
$ ls -lZ
drwxrwx---  root     secret   system_u:object_r:default_t:s15  secret
$ cd secret/
$ echo "file1: First line" > file1
$ echo "file2: First line" > file2
$ ls -lZ
-rw-rw-r--  secret   secret   secret:object_r:default_t:s15    file1
-rw-rw-r--  secret   secret   secret:object_r:default_t:s15    file2
$ chcon system_u:object_r:default_t:s15 file2
$ ls -lZ file2
-rw-rw-r--  secret   secret   system_u:object_r:default_t:s15  file2
$ cat file1 file2
file1: First line
file2: First line
```

```
$ echo "file1: Second line" >> file1
$ echo "file2: Second line" >> file2
$ ls -lZ
-rw-rw-r--  secret   secret   secret:object_r:default_t:s15    file1
-rw-rw-r--  secret   secret   system_u:object_r:default_t:s15  file2
# (create a directory)
$ mkdir testdir1
$ ls -lZd testdir1/
drwxrwxr-x  secret   secret   secret:object_r:default_t:s15    testdir1/
```

As can be seen, the user `secret`, is now able to read information about the subdirectory `secret` and perform tasks such as creating files and adding contents to them and creating new subdirectories. Also note that the `DAC` security mechanisms are utilised since the only users able to access the subdirectory `secret` is root and any user present in the group `secret`. Moreover, the security context of the file "file2" is changed to have the user portion of it to contain the generic system user `system_u`, this is done to compare accesses that are allowed by the policy for the use hidden in the next section.

Also note that the files, and the directory, created by the user `secret`, all have the `DAC` portion mechanism flags set in such a way that the members of the group `secret` may freely modify them.

To see one of the potential problems with using MLS under SELinux, the working directory is changed back into the home directory, `/home/secret`, and a few operations are performed on the file "afile" that was created in the beginning of this test.

```
$ cd /home/secret
$ cat afile
A line
$ echo "A line" > afile
bash: afile: Access denied
$ ls -lZd .
drwxr-xr-x  secret   secret   secret:object_r:user_home_dir_t:s0
```

Since the policy is active, and it enforces the strict Bell–Lapdula MLS policy, the user `secret` is able to read the file, but not write to it (cf. Figure 9.1 and Section 6.4.1 for a description of the "no write down" property).

The privileged user hidden

The second privileged user, `hidden`, is used in a manner similar to that of the user `secret`.

First, the initial security level of the user is displayed, and the file "afile" in `secret`'s home directory is read.

```
# enforce is on
$ whoami
hidden
$ id -Z
hidden:system_r:unconfined_t:s0-s15:c0.c255
$ pwd
/home/hidden
$ ls -lZ /home/secret/afile
-rw-rw-r--  secret   secret   secret:object_r:user_home_t:s0   /home/secret/afile
$ cat /home/secret/afile
A line
```

The working directory is changed to the repository in `/internal/`. Just as for `secret`, that directory is fully accessible to the `hidden`, while access to the subdirectory `secret` is denied.

```
$ cd /internal
$ ls -lZd .
drwxr-xr-x  root     root     system_u:object_r:default_t:s0   .
$ cd secret/
bash: cd: secret/: Access denied
$ ls -lZ
?---------  ?        ?                                        secret
```

The policy enforcement is deactivated to give the user `hidden` the ability to update the security level.

```
# enforce is off
$ newrole -l s15
Authenticating hidden.
Password: <password>
$ id -Z
hidden:system_r:unconfined_t:s15-s15:c0.c255
```

Setting the security level to `s15`, enables the user `hidden` to access the subdirectory `secret`. The working directory is changed to that subdirectory, the contents of the two files "file1" and "file" is read and the subdirectory "testdir1/" is entered.

```
# enforce is on
$ cd secret/
$ ls -lZ
-rw-rw-r--  secret   secret   secret:object_r:default_t:s15    file1
-rw-rw-r--  secret   secret   system_u:object_r:default_t:s15  file2
drwxrwxr-x  secret   secret   secret:object_r:default_t:s15    testdir1
$ cat file1 file2
file1: First line
file1: Second line
file2: First line
file2: Second line
$ cd testdir1
$ pwd
/internal/secret/testdir1
```

From the tests done with the user `hidden`, the conclusion that users at the same level of authorization is able to access the same files under the policy configuration.

The unprivileged user normal

To test the protection of the contents of the repository against access attempts made by unauthorized users, the user `normal` is used to try to access the repository directory `/internal/secret/`.

```
$ whoami
normal
$ id -Z
user_u:system_r:unconfined_t:s0-s14:c0.c255
$ cd /internal
$ ls -lZ
?---------  ?        ?                                        secret
$ cd secret/
bash: cd secret/: Access denied
```

As is obvious from the output of the attempt the directory to the `secret` subdirectory, as well as accessing the subdirectory information, is prevented by the policy enforcement mechanisms.

9.4.4 Problems with the policy implementation

The policy implementation has one big problem that is not readily apparent; the security level `s15` is the highest of the sixteen available security levels. Hence, this security level must be available to at least the user root and the system user that is used to run the `system's` maintenance tools. Moreover, this security level is assigned to other portions of the file system that has been deemed to be very important. Due to this, these `system` users must be implicitly trusted within the policy. It is not possible to create a new security level `s16` using the current security framework, there exists a hard coded limit of sixteen such levels.

9.4.5 Conclusions concerning the policy implementation

The implementation of the strict Bell–Lapdula under the SELinux reference policy is very easy. The author has worked with the older targeted policy configuration, and the interdependencies between the different portions of that policy makes is unnecessarily hard to configure.

Moreover, the tests in Section 9.4.3 established that the policy complies with the specifications established in Section 9.3, and hence the example policy is successful in creating a repository in which secret documents can be secure confined from non–trusted users on a `system`.

Chapter 10

Comparing SELinux' policy language to the policy algebra

In this chapter, the algebra discussed in chapter 7 will be compared to the language used to specify SELinux policies [McC04, Sec] with respect to expressiveness and flexibility. However, it will be done in an informal manner, and hence, differences in expressiveness can only be argued, not proved.

The expressiveness in the areas concerning rights' association with a `subject` or `role`, access restriction, ability to catch errors in the resulting policy and readability is discussed.

10.1 Access restriction

The ability to restrict access is the very purpose of the a `MAC` policy. Hence, this is by far the most important thing to consider; is the language able to express the restrictions that must be imposed?

Both the algebra discussed in Chapter 7 and the SELinux' policy language can assign any (valid) permission to any `object` that may be of interest.

Both languages lack in two different areas though. The algebra has no way to use some form of regular expression to specify a range of files and directories in a concise way. This makes it tedious to assign precise mappings of access rights to all files and directories present on a `system`. The SELinux' configuration on the other hand is restricted by the abilities of the available mechanisms that implement actual security, which, to mention one issue, does not support distributed policies, which may be expressed in the purely symbolic algebra.

Given that the two candidates has very different basis, the result must be that it is a draw, simply due to the fact that both the polices can impose most of the really important restrictions and SELinux' is an actual, working, security mechanism.

10.2 Access right association

Both the algebra and the SELinux' policy languages support the association of rights and access restrictions to both `subject` and `roles`. The policy does this directly, while in SELinux', the rights are associated with a `role`, which in turn is associated with a `subject`. Hence they are equally powerful with the respect to this category.

10.3 Ability to catch errors in the resulting policy

SELinux' policy language includes support for specifying properties that must hold, such as that a certain permission is only assigned to a certain user. The algebra has support for this through both the scoping and provision operators (cf. Section 7.3.2). Hence. both languages must be considered to be equally powerful with respect to this category since both is able to either detect errors or simply filter them out.

10.4 Readability

In actual development, readability becomes a real concern since, if a developer can not understand some part of, or the whole, policy specification, aforementioned developer will not be able to enhance the policy in ways that may desirable.

SELinux' policy language in it self is rather easy to read, at least after some practice. One of the things that makes the language easy to read is that operators are made up of real words, with self explaining names. Moreover, more and more of the actual policy is implemented using macros with self explaining names, and the macros are well documented. The macros themselves are built using a rather easy to read macro language called `m4` to generate more complex statements in the actual policy language.

The common algebra discussed in Chapter 7 on the other hand uses symbols from logic and set theory to express a policy. Although powerful, the readability suffer very badly from this. The biggest issue though is the fact that the syntax for the scoping and provision operator is too alike, making it hard to remember exactly which one is which if not working with the algebra on a fairly regular basis (cf. Section 7.3.2 for a description of the syntax).

Due to the fact the SELinux' policy specification is made using a mixture of well documented macros with, reasonably, well chosen names ,and that said macros and the language has been documented, that configuration language must be said to be the stronger candidate with respect to readability.

10.5 Conclusion

Both the algebra from Chapter 7 and the SELinux' policy language are of comparable strength with respect to expressiveness. Furthermore, both have the capability to catch configuration errors. But the fact that SELinux' policy language has a much higher readability than the algebra makes it the overall winner in this comparison. Readability is simply a very important issue in the real world.

However, the algebra's strength lies in that it is independent of any security mechanism, making it a very strong candidate for expressing policies that are to be used on several different `systems`, with potentially different configuration environments.

Chapter 11

Conclusions

The goals for this thesis were to examine `Mandatory Access Control` using both a theoretical and a practical approach. The main purpose of the theoretical approach was to discuss an algebra that was expressive enough to describe restrictions and assignments of access rights at an abstract level. The main goal of the practical approach was to discuss the SELinux security framework. Using the discussed theory and technology, the two approaches were to be combined in the formulation and implementation of a simple security policy; the policy was to be specified using the algebra and implemented in SELinux. The then two approaches was to be compared with respect to expressiveness.

All of the goals set up for the thesis were achieved. The introductory theory presented in the first chapters of the thesis provided the necessary components to discuss the algebra in Chapter 7, which was then used, together with a simple extension based on basic logic, to define a MLS security policy. The policy was implemented in the SELinux security framework discussed in Chapter 8, using the relatively new reference policy configuration. The policy was tested in Chapter 9, which established that the policy performed correctly. The two approaches were compared in Chapter 10. The comparison established that the policy language was slightly stronger than that of the algebra due to a much higher level of readability.

If one should try to predict the direction of any continued efforts, in the case of the SELinux community and its development efforts, it will most probably concern modularity of the policy and support for distributed policies. This is important, since a modular policy will enable `system` administrators to load and unload specific parts of he policy without the need to recompile the whole policy. Distributed policy enforcement on the other hand is important for implementing a uniform policy that is enforced in a networked environment. For example, that will make it possible to ensure the protection of documents on a corporate network. Moreover, it seems that more powerful policy languages, that removes the need for a separate macro language, seems to be under development.

Chapter 12

Acknowledgements

I would like to thank my supervisor Jonny Pettersson for his patience with my somewhat slow progress at times and for his advise on matters where I got stuck. I would also like to thank Frank Drewes for his help with the problems I had with the algebra.

References

[Amo94] Edward Amoroso. *Fundamentals of Computer Security Technology.* Prentice Hall, 1994.

[And01a] Ross Andersson. *Security Engineering - A Guide to Building Dependable Distribution Systems.* John Wiley & Sons, Inc., 2001.

[And01b] Andrew S. Tanenbaum. *Modern Operation Systems, 2nd Edition.* Prentice Hall, 2001.

[Bis03] Matt Bishop. *Computer Security, Art and Science.* Addison Wesley, 2003.

[Bro] Brown Biomed department. Robotic Surgery. http://biomed.brown.edu/Courses/BI108/ BI108_2005_Groups/04/index.html (visited 2005-11-14).

[CSR] CSRC. Security and Planning in the Computer System Life Cycle. http://csrc.nist.gov/publications/nistpubs/800-12/800-12-html/chapter8.html (visited 2005-11-15).

[Dag98] Dag Ingvar Jacobsen and Jan Thorsvik. *Hur moderna organisationer fungerar.* Studentlitteratur, 1998. Written in swedish.

[Dum03a] Duminda Wijesekera and Sushil Jajodia. A Propsitional Policy Algera for Access Control. *ACM Transactions on Information and Systems Security*, 6 (2):286–325, May 2003.

[Dum03b] Duminda Wijesekera and Sushil Jajodia. Policy Algebras for Access Control - The Propsitional Case. *Proceedings of the 8th ACM conference on Computer and Communications Security*, pages 38–47, November 2003.

[Fed] Fedora Project. SELinux FAQs. http://fedora.redhat.com/docs/selinux-faq/ (visited 2005-12-14).

[Har76] Harrison, Ruzzo and Ullman. Protection in Operating Systems. *Communications of the ACM*, 19 (8):461–471, 1976.

[IAC] IACS. Common Criteria levels. http://www.cesg.gov.uk/site/iacs/index.cfm?menuSelected=1&displayPage=13 (visited 2005-11-15).

[Ken96] Kenneth M. Walker, Daniel F. Sterne, M. Lee Badger, Michael J Petkac, David L. Shermann and Karen A. Oostendorp. Confining Root Programs with Domain and Type Enforcement (DTE). *Proceedings of the Sixth USENIX UNIX Security Symposium, San Jose, California*, 1996.

[Lee95] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker and Sheila A. Haghighat. A Domain and Type Enforcement UNIX Prototype. *Proceedings of the Fifth USENIX UNIX Security Symposium, Salt Lake City, Utah*, 1995.

[Lin] Linux Programmer's Manual. NETLINK (7). http://www.linuxinfor.com/english/man7/netlink.html (visited 2005-11-14).

[Mar03] Mark G. Graff and R. van Wyk. *Secure Coding, Principels and Practices.* O'Reilly, 2003.

[McC04] Bill McCarty. *SELinux – NSA's Open Source Security Enhanced Linux.* O'Reilly, 2004.

[Mor01] Mordechai Ben-Ari. *Mathematical Logic for Computer Science.* Springer, 2001.

[Nov] Novell. SuSe Linux 10.0. http://www.novell.com/products/suselinux/security.html (visited 2006-01-23).

[Ope] OpenBSD project. OpenBSD Security. http://openbsd.org/security.html (visited: 2005-12-05).

[Peta] Peter A. Loscocco, NSA and Stephen D. Smalley, NAI Labs. Meeting Critical Security Objectives with Security Enhanced Linux. http://www.nsa.gov/selinux/papers/ottawa01-abs.cfm (visited 2005-11-15).

[Petb] Peter A. Loscocco, Stephen D. Smalley, Patrick A. Muckelbauer, Ruth C. Taylor, S. Jeff Turner and John F. Farrell. The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments. http://www.nsa.gov/selinux/papers/inevit-abs.cfm (visited 2005-11-15).

[Petc] Peter Loscocco and Stephen Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. http://www.nsa.gov/selinux/papers/freenix01-abs.cfm (visited 2005-11-15).

[pla] Planet SELinux. http://www.selinuxnews.org/planet (visited: 2006-03-19).

[Pro] GNU Project. GNU m4. http://www.gnu.org/software/m4 (visited: 2006-03-19).

[Ral00] Ralph P. Grimaldi. *Discrete and Combinatorial Mathematics.* Addison Wesley Longman, April 2000.

[Rav88] Ravinderpal Singh Sandhu. The Schematic Protection Problem: Its Definition and Analysis for Acyclic Attenuating Schemes. *Communications of the ACM*, 35 (2):404–432, 1988.

[Rav96] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein and Charles E. Youman. Role-Based Access Control Models. *IEEE Computer*, 29 (2):38–47, 1996.

[Ray] Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, David Andersen and Jay Lepreau. The Flask Architecture: System Support for Diverse Security Policies. http://www.nsa.gov/selinux/papers/flask-abs.cfm (visited 2005-11-15).

[Sal] Salim,Khosravi, Kleen and Kuznetsov. Linux Netlink as an IP Services Protocol. http://rfc3549.x42.com/ (visited 2005-11-14).

[Sch00] Bruce Schneier. *Secrets and Lies.* John Wiley & Sons, Inc., 2000.

[Sec] Security Enhanced Linux Reference Policy Project. Security Enhanced Linux Reference Policy. http://serefpolicy.sourceforge.net/index.php (visited: 2006-02-24).

[Sha] Shari L. Phleeger. *Software Engineering: Theory and Practice, 2nd Edition.* Prentice Hall.

[Stea] Stephen Smalley. Configuring the SELinux Policy. http://www.nsa.gov/selinux/papers/policy2-abs.cfm (visited 2005-12-14).

[Steb] Stephen Smalley, Chris Vance and Wayne Salamon. Implementing SELinux as a Linux Security Module. http://www.nsa.gov/selinux/papers/module-abs.cfm (visited 2005-11-15).

[Sud97] Thomas A. Sudkamp. *Languages and Machines, An Introduction to the Theory of Computer Science. Second Edition.* Addison Wesley, 1997.

[Syl00] Sylvia Osborn, Ravi S. Sandhu and Qamar Munawer. Configuring Role-Based Access Control to Enforce Mandatory and Discretionary Access Control Policies. *ACM Transactions on Information and Systems Security*, 3 (2):85–106, 2000.

[Sys] Systrace Policy Generation. http://www.systrace.org (visited 2006-01-23).

[tre] Tresys Technology. http://www.tresys.com (visited: 2006-03-19).

Appendix A

Source Code

The files modified in Section 9.4.2 to alter the standard configuration of the SELinux reference configuration are included for reference. All changes have comments beginning with `@DIFF` before they occur. The simple application security configuration has been excluded due to the trivial changes, which are explained in Section 9.4.2.

A.1 Compilation configuration

Options used to configure the compilation process for the policy configuration (i.e. the file `/etc/selinux/refpolicy/src/policy/build.conf`).

```
########################################
#
# Policy build options
#

# Policy version
# By default, checkpolicy will create the highest
# version policy it supports.  Setting this will
# override the version.  This only has an
# effect for monolithic policies.
#OUTPUT_POLICY = 18

# Policy Type
# strict, targeted,
# strict-mls, targeted-mls,
# strict-mcs, targeted-mcs
# @DIFF TYPE = strict
# @DIFF Note the differencies in the configuration in
# @DIFF file "policy/users"
TYPE = targeted-mls

# Policy Name
# If set, this will be used as the policy
# name.  Otherwise the policy type will be
# used for the name.
NAME = refpolicy
```

```
# Distribution
# Some distributions have portions of policy
# for programs or configurations specific to the
# distribution.  Setting this will enable options
# for the distribution.
# redhat, gentoo, debian, and suse are current options.
# Fedora users should enable redhat.
# @DIFF DISTRO = redhat
DISTRO = redhat

# Direct admin init
# Setting this will allow sysadm to directly
# run init scripts, instead of requring run_init.
# This is a build option, as role transitions do
# not work in conditional policy.
# @DIFF DIRECT_INITRC=n
DIRECT_INITRC=y

# Build monolithic policy.  Putting n here
# will build a loadable module policy.
MONOLITHIC=y

# Polyinstantiation
# Enable polyinstantiated directory support.
POLY=n

# Uncomment this to disable command echoing
#QUIET:=@
```

A.2 Home directory security context configuration

The generated home directories security context configuration (i.e. the file `/etc/selinux/refpolicy/contexts/files/file_contexts.homedirs`.

```
#
#
# User-specific file contexts, generated via /usr/sbin/genhomedircon
# edit /etc/selinux/refpolicy/users/local.users to change file_context
#
#



#
# Home Context for user user_u
#

/home/[^/]*/.+ user_u:object_r:user_home_t:s0
/home/[^/]*/.*/plugins/libflashplayer\.so.* --user_u:object_r:textrel_shlib_t:s0
/home/[^/]*/((www)|(web)|(public_html))(/.+)? user_u:object_r:httpd_user_content_t:s0
/home/[^/]* -d user_u:object_r:user_home_dir_t:s0
/home/a?quota\.(user|group) --system_u:object_r:quota_db_t:s0
/home/lost\+found/.* <<none>>
```

```
# @DIFF /home -d system_u:object_r:home_root_t:s15:c0.c255
/home -d user_u:object_r:user_home_dir_t:s0
/home/\.journal <<none>>
/home/lost\+found -d system_u:object_r:lost_found_t:s15:c0.c255


#
# Other Context for user .*
#





#
# Home Context for user secret
#

/home/secret/.+ secret:object_r:user_home_t:s0
/home/secret/.*/plugins/libflashplayer\.so.* --secret:object_r:textrel_shlib_t:s0
/home/secret/((www)|(web)|(public_html))(/.+)? secret:object_r:httpd_user_content_t:s0
/home/secret -d secret:object_r:user_home_dir_t:s0


#
# Other Context for user secret
#





#
# Home Context for user root
#

/root/.+ root:object_r:user_home_t:s0
/root/.*/plugins/libflashplayer\.so.* --root:object_r:textrel_shlib_t:s0
/root/((www)|(web)|(public_html))(/.+)? root:object_r:httpd_user_content_t:s0
/root -d root:object_r:user_home_dir_t:s0


#
# Other Context for user root
#





#
# Home Context for user hidden
#

/home/hidden/.+ hidden:object_r:user_home_t:s0
```

```
/home/hidden/.*/plugins/libflashplayer\.so.* --hidden:object_r:textrel_shlib_t:s0
/home/hidden/((www)|(web)|(public_html))(/.+)? hidden:object_r:httpd_user_content_t:s0
/home/hidden -d hidden:object_r:user_home_dir_t:s0


#
# Other Context for user hidden
#
```

A.3 File system security context configuration

The file system security context configuration. Note the configuration for the `/internal` directory and its subdirectories (i.e. the file `/etc/selinux/refpolicy/src/policy/policy/modules/kernel/files.fc`).

```
#
# /
#
/.* gen_context(system_u:object_r:default_t,s0)
/ -d gen_context(system_u:object_r:root_t,s0)
/\.journal <<none>>

ifdef(‘distro_redhat’,‘
/\.autofsck --gen_context(system_u:object_r:etc_runtime_t,s0)
/\.autorelabel --gen_context(system_u:object_r:etc_runtime_t,s0)
/fastboot  --gen_context(system_u:object_r:etc_runtime_t,s0)
/forcefsck  --gen_context(system_u:object_r:etc_runtime_t,s0)
/fsckoptions  --gen_context(system_u:object_r:etc_runtime_t,s0)
/halt --gen_context(system_u:object_r:etc_runtime_t,s0)
/poweroff --gen_context(system_u:object_r:etc_runtime_t,s0)
’)

ifdef(‘distro_suse’,‘
/success --gen_context(system_u:object_r:etc_runtime_t,s0)
’)

# @DIFF
# @DIFF /internal
# @DIFF All under "secret" is to be protected
# @DIFF
/internal -d gen_context(system_u:object_r:default_t,s0)
/internal/secret -d   gen_context(system_u:object_r:default_t,s15)
/internal/secret(/.*)?    gen_context(system_u:object_r:default_t,s15)

#
# /boot
#
/boot/\.journal <<none>>
/boot/lost\+found -d gen_context(system_u:object_r:lost_found_t,s15:c0.c255)
```

```
/boot/lost\+found/.* <<none>>

#
# /emul
#

ifdef(‘distro_redhat’,‘
/emul(/.*)? gen_context(system_u:object_r:usr_t,s0)
’)

#
# /etc
#
/etc(/.*)? gen_context(system_u:object_r:etc_t,s0)
/etc/\.fstab\.hal\..+ --gen_context(system_u:object_r:etc_runtime_t,s0)
/etc/asound\.state --gen_context(system_u:object_r:etc_runtime_t,s0)
/etc/blkid\.tab.* --gen_context(system_u:object_r:etc_runtime_t,s0)
/etc/fstab\.REVOKE --gen_context(system_u:object_r:etc_runtime_t,s0)
/etc/HOSTNAME --gen_context(system_u:object_r:etc_runtime_t,s0)
/etc/ioctl\.save --gen_context(system_u:object_r:etc_runtime_t,s0)
/etc/issue --gen_context(system_u:object_r:etc_runtime_t,s0)
/etc/issue\.net --gen_context(system_u:object_r:etc_runtime_t,s0)
/etc/localtime -l gen_context(system_u:object_r:etc_t,s0)
/etc/mtab --gen_context(system_u:object_r:etc_runtime_t,s0)
/etc/motd --gen_context(system_u:object_r:etc_runtime_t,s0)
/etc/nohotplug --gen_context(system_u:object_r:etc_runtime_t,s0)
/etc/nologin.* --gen_context(system_u:object_r:etc_runtime_t,s0)
/etc/smartd\.conf --gen_context(system_u:object_r:etc_runtime_t,s0)

/etc/cups/client\.conf --gen_context(system_u:object_r:etc_t,s0)

/etc/init\.d/functions --gen_context(system_u:object_r:etc_t,s0)

/etc/ipsec\.d/examples(/.*)? gen_context(system_u:object_r:etc_t,s0)

/etc/network/ifstate --gen_context(system_u:object_r:etc_runtime_t,s0)

/etc/ptal/ptal-printd-like --  gen_context(system_u:object_r:etc_runtime_t,s0)

/etc/rc\.d/init\.d/functions -- gen_context(system_u:object_r:etc_t,s0)

/etc/sysconfig/hwconf --gen_context(system_u:object_r:etc_runtime_t,s0)
/etc/sysconfig/iptables\.save -- gen_context(system_u:object_r:etc_runtime_t,s0)
/etc/sysconfig/firstboot --gen_context(system_u:object_r:etc_runtime_t,s0)

ifdef(‘distro_gentoo’, ‘
/etc/profile\.env --gen_context(system_u:object_r:etc_runtime_t,s0)
/etc/csh\.env --gen_context(system_u:object_r:etc_runtime_t,s0)
/etc/env\.d/.* --gen_context(system_u:object_r:etc_runtime_t,s0)
’)

ifdef(‘distro_redhat’,‘
/etc/rhgb(/.*)? -d gen_context(system_u:object_r:mnt_t,s0)
')

ifdef(`distro_suse',`
/etc/defkeymap\.map --gen_context(system_u:object_r:etc_runtime_t,s0)
/etc/init\.d/\.depend.* --gen_context(system_u:object_r:etc_runtime_t,s0)
')

#
# HOME_ROOT
# expanded by genhomedircon
#
HOME_ROOT -d gen_context(system_u:object_r:home_root_t,s15:c0.c255)
HOME_ROOT/\.journal <<none>>
HOME_ROOT/lost\+found -d gen_context(system_u:object_r:lost_found_t,s15:c0.c255)
HOME_ROOT/lost\+found/.* <<none>>

#
# /initrd
#
# initrd mount point, only used during boot
/initrd -d gen_context(system_u:object_r:root_t,s0)

#
# /lost+found
#
/lost\+found -d gen_context(system_u:object_r:lost_found_t,s15:c0.c255)
/lost\+found/.* <<none>>

#
# /media
#
# Mount points; do not relabel subdirectories, since
# we don't want to change any removable media by default.
/media(/[^/]*)? -d gen_context(system_u:object_r:mnt_t,s0)
/media/[^/]*/.* <<none>>

#
# /mnt
#
/mnt(/[^/]*)? -d gen_context(system_u:object_r:mnt_t,s0)
/mnt/[^/]*/.* <<none>>

#
# /opt
#
/opt(/.*)? gen_context(system_u:object_r:usr_t,s0)

/opt(/.*)?/var/lib(64)?(/.*)? gen_context(system_u:object_r:var_lib_t,s0)

#
# /proc
#
/proc(/.*)?                     <<none>>
```

```
#
# /selinux
#
/selinux(/.*)?                  <<none>>

#
# /srv
#
/srv(/.*)? gen_context(system_u:object_r:var_t,s0)

#
# /sys
#
/sys(/.*)?                      <<none>>

#
# /tmp
#
/tmp -d gen_context(system_u:object_r:tmp_t,s0-s15:c0.c255)
/tmp/.* <<none>>
/tmp/\.journal <<none>>

/tmp/lost\+found -d gen_context(system_u:object_r:lost_found_t,s15:c0.c255)
/tmp/lost\+found/.* <<none>>

#
# /usr
#
/usr(/.*)? gen_context(system_u:object_r:usr_t,s0)
/usr/\.journal <<none>>

/usr/etc(/.*)? gen_context(system_u:object_r:etc_t,s0)

/usr/inclu.e(/.*)? gen_context(system_u:object_r:usr_t,s0)

/usr/local/\.journal <<none>>

/usr/local/etc(/.*)? gen_context(system_u:object_r:etc_t,s0)

/usr/local/lost\+found -d gen_context(system_u:object_r:lost_found_t,s15:c0.c255)
/usr/local/lost\+found/.* <<none>>

/usr/local/src(/.*)? gen_context(system_u:object_r:src_t,s0)

/usr/lost\+found -d gen_context(system_u:object_r:lost_found_t,s15:c0.c255)
/usr/lost\+found/.* <<none>>

/usr/share(/.*)?/lib(64)?(/.*)? gen_context(system_u:object_r:usr_t,s0)

/usr/src(/.*)? gen_context(system_u:object_r:src_t,s0)

/usr/tmp -d gen_context(system_u:object_r:tmp_t,s0-s15:c0.c255)
```

```
/usr/tmp/.* <<none>>

#
# /var
#
/var(/.*)? gen_context(system_u:object_r:var_t,s0)
/var/\.journal <<none>>

/var/db/.*\.db --gen_context(system_u:object_r:etc_t,s0)

/var/ftp/etc(/.*)? gen_context(system_u:object_r:etc_t,s0)

/var/lib(/.*)? gen_context(system_u:object_r:var_lib_t,s0)

/var/lib/nfs/rpc_pipefs(/.*)? <<none>>

/var/lock(/.*)? gen_context(system_u:object_r:var_lock_t,s0)

/var/lost\+found -d gen_context(system_u:object_r:lost_found_t,s15:c0.c255)
/var/lost\+found/.* <<none>>

/var/run -d gen_context(system_u:object_r:var_run_t,s0-s15:c0.c255)
/var/run/.* gen_context(system_u:object_r:var_run_t,s0)
/var/run/.*\.*pid <<none>>

/var/spool(/.*)? gen_context(system_u:object_r:var_spool_t,s0)

/var/tmp -d gen_context(system_u:object_r:tmp_t,s0-s15:c0.c255)
/var/tmp/.* <<none>>
/var/tmp/lost\+found -d gen_context(system_u:object_r:lost_found_t,s15:c0.c255)
/var/tmp/lost\+found/.* <<none>>
/var/tmp/vi\.recover -d gen_context(system_u:object_r:tmp_t,s0)
```

A.4 User configuration

Configuration of security levels available to users (i.e. the file `/etc/selinux/refpolicy/src/policy/policy/users`).

```
##################################
#
# Core User configuration.
#

#
# gen_user(username, role_set, mls_defaultlevel, mls_range, [mcs_catetories])
#

#
# system_u is the user identity for system processes and objects.
# There should be no corresponding Unix user identity for system,
# and a user process should never be assigned the system user
# identity.
```

```
#
# @DIFF Note that system_u is considered a "trustworthy" user;
# @DIFF system_u can read level s15 files
gen_user(system_u, system_r, s0, s0 - s15:c0.c255, c0.c255)

#
# user_u is a generic user identity for Linux users who have no
# SELinux user identity defined.  The modified daemons will use
# this user identity in the security context if there is no matching
# SELinux user identity for a Linux user.  If you do not want to
# permit any access to such users, then remove this entry.
#
# @DIFF  ifdef(‘targeted_policy’,‘
# @DIFF  gen_user(user_u, user_r sysadm_r system_r, s0, s0 - s15:c0.c255, c0.c255)
# @DIFF  ’,‘
# @DIFF  gen_user(user_u, user_r, s0, s0 - s15:c0.c255, c0.c255)
# @DIFF  ’)
# @DIFF
# @DIFF Upper sensitivity level is for "secret" only: s/s15/s14
ifdef(‘targeted_policy’,‘
gen_user(user_u, user_r sysadm_r system_r, s0, s0 - s14:c0.c255, c0.c255)
’,‘
gen_user(user_u, user_r, s0, s0 - s14:c0.c255, c0.c255)
’)


# @DIFF The "secret" user, will have access to sensitivity level s15
# @DIFF and have default level s0 to enable currect login (the files
# @DIFF $HOME must be of level s0 to work correctly, and files on that
# @DIFF level can not be written to under BLP)
gen_user(secret, user_r system_r, s0, s0 - s15:c0.c255, c0.c255)

# @DIFF The "hidden" user, will have access to sensitivity level s15
# @DIFF and have default level s0 to enable currect login (the files
# @DIFF $HOME must be of level s0 to work correctly, and files on that
# @DIFF level can not be written to under BLP)
# @DIFF Used to demonstrate what must be done to share files
gen_user(hidden, user_r system_r, s0, s0 - s15:c0.c255, c0.c255)

#
# The following users correspond to Unix identities.
# These identities are typically assigned as the user attribute
# when login starts the user shell.  Users with access to the sysadm_r
# role should use the staff_r role instead of the user_r role when
# not in the sysadm_r.
#
# @DIFF Note that root is considered a "trustworthy" user; root can
# @DIFF read level s15 files
ifdef(‘targeted_policy’,‘
gen_user(root, user_r sysadm_r system_r, s0, s0 - s15:c0.c255, c0.c255)
’,‘
ifdef(‘direct_sysadm_daemon’,‘
gen_user(root, sysadm_r staff_r system_r, s0, s0 - s15:c0.c255, c0.c255)
',`
gen_user(root, sysadm_r staff_r, s0, s0 - s15:c0.c255, c0.c255)
')
')
```