Matias Fallenius

Comparison of Security Models and Security Policy Languages

Master's Thesis
2nd December 2004

HELSINKI UNIVERSITY OF TECHNOLOGY
Department of Computer Science and Engineering
Telecommunications Software and Multimedia Laboratory

HELSINKI UNIVERSITY OF TECHNOLOGY

ABSTRACT OF THE MASTER'S THESIS

Author:	Matias Fallenius
Title:	Comparison of Security Models and Security Policy Languages
Date:	2nd December 2004
Pages:	108
Department:	Department of Computer Science and Engineering
Professorship:	T-110
Supervisor:	Professor Teemupekka Virtanen

Most software applications are designed and developed without an explicitly defined security model. Rather security features are often implemented as afterthoughts on an ad-hoc basis. There exist various formally defined security models that are similar in many but not all aspects. The primary objective of this thesis is to study and compare a set of currently available security models and security policy languages and to find out the features that are common to all of them. The secondary objective is to briefly assess their applicability to serve as general-purpose security models and policy languages in a policy-based security architecture.

Keywords:	Computer security, Information security, Security model, Security policy, Security policy language
Language:	English

TEKNILLINEN KORKEAKOULU DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Matias Fallenius
Työn nimi:	Tietoturvamallien ja tietoturvapolitiikkakielten vertailu
Päivämäärä:	2. joulukuuta 2004
Sivumäärä:	108
Osasto:	Tietotekniikan osasto
Professuuri:	T-110
Työn valvoja:	Professori Teemupekka Virtanen

Suurin osa tietokoneohjelmistoista suunnitellaan ja toteutetaan ilman eksplisiittisesti määriteltyä tietoturvamallia. Usein tietoturvaominaisuudet toteutetaan suunnittelemattomasti ja vasta jälkikäteen. On olemassa useita eri formaalisti määriteltyjä tietoturvamalleja, joilla on paljon samoja piirteitä mutta myös merkittäviä eroavaisuuksia. Tämän diplomityön päätavoite on tutkia eräitä tietoturvamalleja ja tietoturvapolitiikoita kuvaavia kieliä ja löytää ne ominaisuudet, jotka ovat yhteisiä kaikille malleille. Toissijainen tavoite on arvioida eri mallien ja kielten soveltuvuutta osaksi politiikkoihin perustuvaa tietoturva-arkkitehtuuria.

Avainsanat:	Tietoturva, Tietoturvamalli, Tietoturvapolitiikka, Tietoturvapolitiikkakieli
Kieli:	Englanti

Preface

Computer security is painfully hard to get right! The very programs and systems that we all use abound with critical security bugs. But it has always been like that. It is just that recently the Internet has connected all computers together. Now buggy or intentionally harmful software can cause much more disastrous results. Consequently, computer security is finally starting to get the attention it has always been craving for. Its importance is getting increasingly recognized also by the general public since the media serves us a dose of security advisories almost daily.

At present, computer security is still an evolving art and not an exact science. This thesis studies one particular area of secure software design, security modeling, and assesses the capability of different security models to function as a building block for secure systems in the future.

Acknowledgements

I would like to express my gratitude to all the people that supported me in writing this thesis both in Italy and in Finland. Especially I thank Professor Antonio Lioy and M.Sc. Cataldo Basile of Politecnico di Torino for providing me the initial topic and for their valuable feedback. Also, I thank Professor Teemupekka Virtanen of Helsinki University of Technology for supervising this thesis.

Contents

List of Figures

List of Abbreviations

API	Application Programming Interface
BLP	Bell-LaPadula
B-N	Brewer-Nash
BST	Basic Security Theorem
CIM	Common Information Model
CNF	Conjunctive Normal Form
C-W	Clark-Wilson
DAC	Discretionary Access Control
DEN	Directory Enabled Networking
DMI	Desktop Management Interface
DMTF	Distributed Management Network
DNF	Disjunctive Normal Form
GAA-API	Generic Authorization and Access-Control API
GSS-API	Generic Security Service API
HTTP	Hypertext Transfer Protocol
HRU	Harrison-Ruzzo-Ullman
IETF	Internet Engineering Task Force
IT	Information Technology
ICT	Information and Communication Technology
J2SE	Java 2 Standard Edition
MAC	Mandatory Access Control
MLS	Multi-Level Security
MOF	Management Object Format
OCL	Object Constraint Language
OID	Object ID
OO	Object Oriented
PCIM	Policy Core Information Model
PDP	Policy Decision Point
PEP	Policy Enforcement Point

QoS	Quality of Service
RBAC	Role Based Access Control
RFC	Request for Comments
SDK	Software Development Kit
SMBIOS	System Management BIOS
UML	Unified Modeling Language
URL	Uniform Resource Locator
WBEM	Web Based Enterprise Management
XACML	eXtensible Access Control Markup Language
XML	eXtensible Markup Language

Chapter 1

Introduction

This chapter explains the motivation for this thesis and defines its objective. Furthermore, a brief background on security modeling and policy based management is provided.

1.1 Motivation

1.1.1 The imperfection of the present

Users in companies and homes alike are increasingly being harassed by computer security problems. Ubiquitous computing brings with it an increasing number and variety of new viruses, malware, spyware, trojan horses, Internet worm attacks, unsolicited email, and so on. A whole lot of new potential applications in areas such as e-commerce and e-government is also prevented from getting off the ground because people (justifiably) do not trust the security of current computer systems.

As computing and network connectivity become increasingly commonplace and widespread, computer security starts to gain importance. Nowadays it is common to see computer security advisories discussed even in the major non-technical newspapers and television news. It seems that time is now finally getting ripe for taking security seriously. We can expect big industry players to really invest in securing their products.

However, currently the computer security field is still in its infancy. As security has not been considered of much importance before, it is still common to design and implement security features in an ad-hoc fashion without an explicit security design. This is hazardous since even minuscule implementation errors can translate to dramatic harm for businesses and consumers.

Security is a complex whole that is affected by technology, business, organizational and people issues. Skills are needed in all these disciplines to design an effective security system and consequently it is difficult to find people with all the required skills. Also, traditionally security has been only seen as a pure cost with no direct impact on revenues. Therefore it is difficult to get any management support before a real disaster takes place.

All security-aware software necessarily has a *security model*. It is a description of all security-related concepts and their interrelationships in the user organization that are to be somehow automatically processed by the software. However, in most cases the security model has not been not explicitly designed. Rather the model takes the form of an ad-hoc gut feeling in the minds of designers and implementers. This approach leads to obvious problems.

Well-defined security models have been designed since the 1970s. There are many models, all of which share the basic objective of facilitating the building a computer system that satisfies the organization's security requirements. However, most security models are tightly coupled to the characteristics of the organizations and computer systems whose security requirements they were designed to satisfy. Thus different models see security problems from their respective and slightly differing viewpoints. Hence, they also come up with different solutions and thus fail to establish common ground. So far, there has not been a security model that would have satisfied the needs of a large number of diverse organizations.

1.1.2 A vision for the future

This lack of common and reusable security models and respective implementations brings about constant reinvention of the wheel every time a new system is designed. The ICT industry, and with its increasing ubiquity also the global society, would greatly benefit from a high-quality, field-proven, flexible security framework. Such a development would lead to more secure and reliable products at lower production costs. It would also accelerate the development of novel and innovative applications of information technology, e.g. e-business, e-government, and collaboration in and between organizations.

1.2 Objective

A commonly accepted general-purpose security model would be one step towards the vision depicted above. However, such a model does not yet exist at the moment. To better understand the requirements for such a model, first the knowledge in the currently available models needs to be analyzed.

Therefore the primary objective of this thesis is to study and compare currently available security models and find out what is common to all of them. The secondary objective is to briefly assess their applicability to serve as general-purpose security models and policy languages in a policy-based security architecture.

1.3 Structure

The currently available security models are presented in Chapter 2. Chapter 3 compares the models and defines the common core. Chapter 4 discusses some of the experiences gathered and concludes the thesis.

1.4 Background

1.4.1 Security modeling

Security began to gain importance as a crucial quality of information systems in the 1960s and 1970s. Early research was initiated and supported mostly by government and defense-related projects.

Security was and continues to be a difficult concept to define. Different parties have different requirements concerning security. The systems that ought to implement the desired security properties are complex and constantly changing. And finally, the weakest link of security is usually the connection between technical systems and human-oriented processes, which are very flexible and hard to define in technological terms.

In order to be able to discuss security this intricate system had to be simplified by constructing models that represented its relevant security-related aspects. Early research concentrated on developing formal mathematical models. It was hoped that by modeling systems formally their security properties could be verified by mathematical proofs. While some notable models were developed and their security could in fact be verified, they proved to be of little use in most real-life situations. This was because in

order to construct models fit for mathematical manipulation a lot of assumptions had to made in abstracting the real world. The assumptions made were too restrictive and simplistic for most use scenarios.

Various formal models were developed which provided valuable information and insight. While the first models were only destined for military environments, research later expanded to commercial environments as well.

Since the early formal models proved to be too restrictive for many real uses, most recent models are less formal, more flexible, and more pragmatic [59]. However, this increased flexibility and informality also means that not much cannot be said about these models by mathematical analysis. In general, these later models focused on discretionary access control decisions made by respective information owners rather than strict mandatory controls enforced by the system itself.

1.4.2 Policy based management

Policy based management has been seen as a viable solution for managing complex distributed systems for some time now [31, 41, 66]. It is applicable to many areas of management, of which most attention has been received by network, Quality of Service, and security management areas.

The goal of policy based management is to view and specify policies in high level abstract terms and automatically refine these to low level technical policies and configuration that satisfy the higher level requirements. This approach has several advantages. Policies can be written in high level terms without knowledge of technological details. This is very useful in large organizations where new technologies are adopted at a quick pace and there is a lack of competent personnel for administering them. Given standardized management interfaces, policies can be automatically applied to a diversity of managed elements in a vendor independent manner. Policies can be automatically distributed to the managed components instead of individually configuring each component. Policy based systems can automatically react to a dynamic change in computing systems and adjust them appropriately without the need of human administrator intervention.

Traditional methods are neither scalable nor flexible enough to manage increasingly complex and heterogenous distributed systems typical of large enterprises of today. Systems are so complex that it is difficult to get a clear picture of the system and its state or to verify if the current configuration is coherent with high level objectives. Systems are so large that manag-

ing each component individually becomes impossible. Vendor dependent management tools are not able to cooperate in a multi vendor network.

Policy based management is suggested as a solution to overcome these problems. System components such as users, services, and devices are organized into groups and roles as in Role Based Access Control systems [21]. Policies are defined in terms of more abstract concepts as high level policies which are automatically translated, or *refined* into lower level policies and configuration and distributed to the end devices. Algorithms could be developed that would detect conflicts between policies, even if at the moment this is still largely an open research issue. There is also ongoing work, such as the DMTF Common Information Model [18], to standardize in a vendor independent manner the management interfaces provided by various devices and services.

Multilevel policies and refinement

There are many definitions for a policy in the literature. In most of the work, two distinct meanings are recognized. The first meaning is that of a high level policy goal such as a business objective that needs to be satisfied. The second meaning is a lower level policy rule that helps the system make the correct decisions to satisfy the policy goals.

Policies can be specified at various levels of abstraction, ranging from business objectives to service configuration and low level device specific programming languages. There doesn't exist a clear division of policies based on their abstractness [1]. Also the automatic translation may be implemented using several intermediate stages. For example, business objectives could be (probably manually) translated to a generic policy language, which is then translated to a configuration language generic to a certain device type, and this is still refined to device and vendor specific configuration. On the other hand, in some cases the policies could be translated directly into low level configuration. In any case, the general idea is to move from a generic, abstract and business oriented objective specification towards detailed, low level, and technology oriented representation that can be distributed to devices and services. This process is called policy translation or policy refinement.

The ultimate goal of policy based management is to facilitate policy specifications at very high levels of abstraction, such as business level objectives written in English, and then refine them automatically. This would allow policy specifications in familiar terms without expert knowledge on

special policy languages. This is a very ambitious goal and at the moment the research community is nowhere close to it yet. Policies written in a human language are obviously prone to the well-known problems of human language processing encountered in the various areas of Artificial Intelligence research (human language policy analysis has been studied in [39]). Ambiguities and errors in the resulting low level policies would be particularly undesirable in the sensitive area of security management. This problem can be avoided by using a formal policy language that is directly interpretable by computers. However, formal languages are harder to use by humans and will therefore require a well trained policy expert to write them.

Policies can be defined using very diverse concepts, terms, and levels of abstraction. According to [1], different levels of abstraction correspond to points of view of different people in an organization. Policies can also be seen to form hierarchies [67, 41, 31]. Different authors have different ways of dividing the hierarchy to levels. In any case, high level policies deal with abstract business oriented concepts, and low level policies are about low level technological details.

In the literature the process of deriving low level policies from high level ones is called *policy translation*, *policy transformation*, or *policy refinement*. In this work the term policy refinement is used. The objective of policy refinement is to perform such a transformation that low level policies completely implement all the requirements set by the higher level policies (*correctness*), and that the low level policies do not conflict with each other (*consistency*) [13].

In practice, policies are specified by many people, for different areas of management, at various levels of abstraction, and using diverse concepts and terminology. It is obvious that deriving a correct and consistent low level policy set in these conditions is difficult. In fact, automatic refinement as a general problem has not been studied extensively yet [13] [42]. Consequently, current implementations focus on simple scenarios and refinement of lower level policies.

Recent work in policy refinement [61] [4] tries to leverage work in requirements engineering such as [15]. The point here is to see high level policies as goals and use techniques such as refinement patterns and goal regression to gradually refine policies into their lower level counterparts.

Policy conflicts

It is possible to for system administrators to write policies that conflict with each other. Conflicts may be caused by human errors, communication problems between different administrators, and dynamically changing policies over time. It is useful to be able to detect potential conflicts and resolve them. Policy conflicts are covered in more detail in Section 2.4.1.

State of the art

Policy based management is being actively researched by several groups in the academic and industrial worlds. Also most network equipment vendors have tools that support policy based management to various extents.

Ponder is a policy research project at the Imperial College of London. It includes in an integrated development environment a generic language for specifying management and security policies, a policy compiler framework for translating policies, a policy editor with a graphical user interface for easy editing of policies, and a domain browser for viewing and managing components and policies installed in the system.

The IETF Policy Working Group has developed the Security Policy Specification Language for describing low level security policies for use with IPSec. It is tightly related to IPSec and not useful for describing other types of security policies.

IETF and DMTF are collaborating on extending the DMTF Common Information Model with policy based management. While not strictly focused on security, the model will eventually provide security as well as other types of policy. This effort is analyzed in detail in section 2.3.1.

The most important open research areas of policy research are conflict detection and automatic policy refinement. Here much work still needs to be done.

For a more extensive review of policy specifications see [13].

Chapter 2

Presentation of security models

This chapter presents the most significant security models currently available. Section 2.1 explains traditional, mostly formal models developed from the 1970s to early 1990s. Section 2.2 describes two practical security models from late 1990s. The chapter concludes with Sections 2.3 and 2.4 that present the most recent models from the few last years that allow flexible security policies to be defined.

2.1 Traditional formal models

2.1.1 Bell-LaPadula

The Bell-LaPadula or BLP model is the best known single computer security model. It was developed by David Bell and Leonard LaPadula of MITRE Corporation in the early 1970s. The BLP deals solely with confidentiality. Other aspects of security such as integrity or availability are not considered. BLP is a so-called Mandatory Access Control (MAC) model which means that the security policy is unconditionally enforced by the system and the respective owners of information cannot have an influence on it at their discretion.

Actually the work consisted of several different models but it is customary to use the term "Bell-LaPadula model" to refer to a certain set of central properties [40]. In addition, in the literature several different simplifications have been presented [34, 36, 37, 22, 2], varying in details such as the formulation of state security properties and the number of access operations. The original model is documented in [7]. The discussion below is

one possible roundup of the various presentations.

In BLP the system consists of the following:

- A set of subjects S. Subjects represent entities such as system users and computer programs executing on their behalf.
- A set of objects O. Objects represent entities such as documents and system files.
- A set of access operations $A = \{read, write\}$.
- A lattice of security labels $(L, \geq)$

In the original model there were two more access operations: *execute* and *append*. However, they are not as essential and hence not considered here.

Security labels

The security label concept originates from military information processing practices. Each piece of information is classified and tagged with a *sensitivity* level. The traditional sensitivity levels in increasing order of sensitivity are *unclassified*, *confidential*, *secret*, and *top secret*. Similarly, staff members are assigned *clearances* based on their reliability and how extensively their background has been checked. The basic rule is that a person is only allowed to access information with a sensitivity level lower than or equal to his clearance level. This scheme is called *Multi-Level Security* (MLS).

Furthermore, information may be additionally tagged with one or more *codewords*, also known as *categories*. These categories could for example denote that the piece of information belongs to a certain project or department. Sets of categories are also called *compartments*. Based on the need-to-know principle, staff members are allowed access to some of these categories. Then, in addition to the sensitivity level requirement stated above, in order to access a piece of information a person must also have access to all the categories that the information has been tagged with.

A security label of a person or information consists of its sensitivity level and the related set of categories. A security label is said to *dominate* another if its sensitivity level is greater than or equal to the sensitivity level of the other and its category set is a superset of the category set of the other. Mathematically speaking, we have a totally ordered set of sensitivity levels SL and a set of categories C. Security labels are ordered pairs $\{(s, c)|s \in$

$SL, c \in C\}$. A security label $l_1 = (s_1, c_1)$ *dominates* $l_2 = (s_2, c_2)$ if and only if $s_1 \geq s_2$ and $c_2 \subseteq c_1$. The *dominates* relation is then a partial ordering on the set of all possible security labels. Together the set of security labels and the *dominates* relation form a lattice structure. Lattices are used in several other security models as well [58]. Figure 2.1 illustrates a real military lattice that was described by Smith in [62]. It consists of the standard sensitivity levels TS, S, C, U and eight categories A, K, L, Q, W, X, Y, Z.

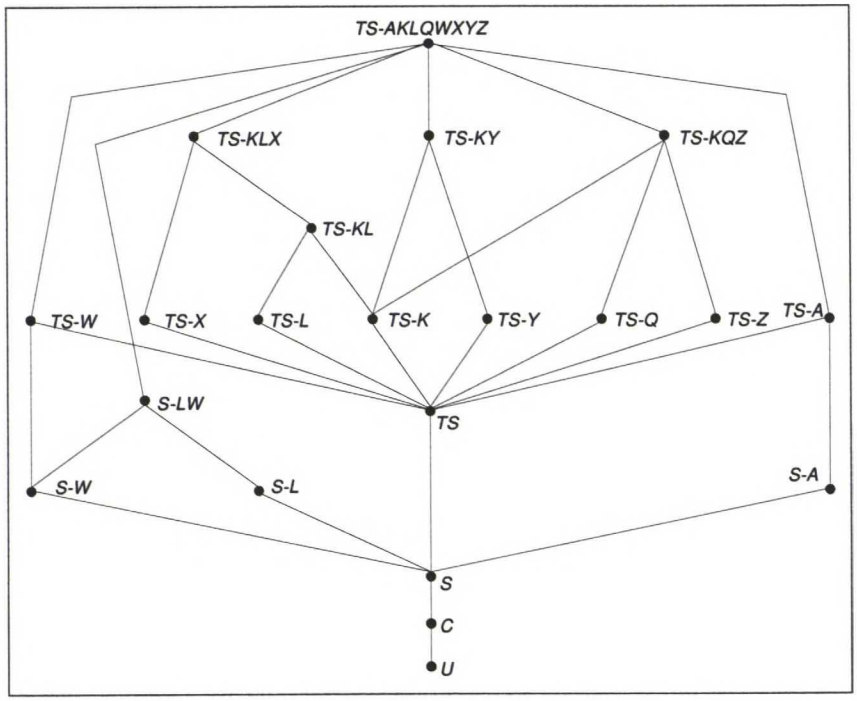

Figure 2.1: Smith's lattice

System states

In BLP the system is modeled as a state machine. A system consists of states and transitions between them. Essentially, a state consists of the following:

- The set of current accesses $B \subseteq \mathcal{P}(S \times O \times A)$.
- The access permission matrix $M = (M_{so})_{s \in S, o \in O}$, $M_{so} \subseteq A$.
- Security label assignment functions $F = (f_s, f_o)$. $f_s : S \to L$ gives the subject's security label, and $f_o : O \to L$ gives the object's security label.

The access permission matrix can be used to impose discretionary restrictions on allowable operations in addition to the mandatory restrictions set by the BLP model itself. The access matrix is a concept originally invented by Lampson [29] and adopted by many other security models as well. There is one row for each subject and one column for each object. Each element in the matrix contains the set of operations the subject is allowed to perform on the object. The storage of the matrix data can be implemented in two principal ways. One way is storing with each object the elements in the corresponding column of the access matrix as a list of (subject, operation) pairs. The columns are called *Access Control Lists* (ACLs). Another way is to attach the row of the matrix to each subject as (object, operation) pairs and is called *Capability Lists*. The access matrix concept is illustrated in Figure 2.2.

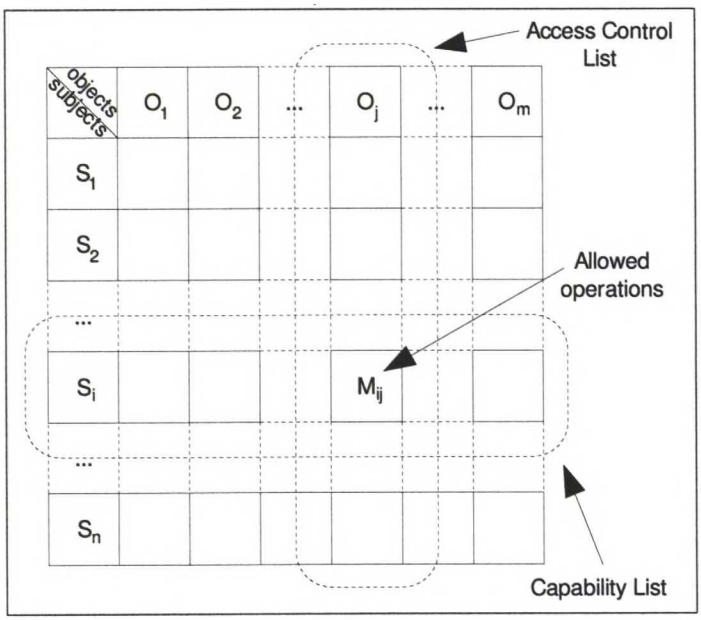

Figure 2.2: Access matrix structure

In the original BLP model, subjects have an additional *current security label* in addition to the standard maximum security label. This allows them to temporarily lower their security clearing in order to for example intentionally limit their access to sensitive material when running untrusted programs or to be able to write to documents with lower security labels (see the security properties of states below).

BLP defines the following three fundamental properties on system states:

- *Simple Security (ss) property* (also known as *No Read Up*):

$$\forall b = (s, o, a) \in B, a = read : f_s(s) \geq f_o(o)$$

  In other words, a state satisfies the ss-property if for all current *read* accesses, the subject's security label dominates the object's security label.

- $\star$*-property* (also known as No Write Down):

$$\forall b = (s, o, a) \in B, a = write : f_o(o) \geq f_s(s)$$

  In other words, a state satisfies the $\star$-property if for all current *write* accesses, the object's security label dominates the subject's current security label.

- *Discretionary Security (ds) property*:

$$\forall b = (s, o, a) \in B : a \in M_{so}$$

  In other words, a state satisfies the ds-property if for all current accesses, the relevant access operation is included in the access permission matrix.

A state is said to be secure if it satisfies all the three properties above.

System security

Being a state machine model, the security of a system is defined through the security of its states. The model's *Basic Security Theorem* (BST) argues that a system is secure if and only if its initial state is secure and all the state transitions are secure. So basically, the BLP-security of a given system can be proved by induction by showing that state transitions only lead to secure states.

The BLP formulation of system security has been a topic of a lot of discussion. In [34] McLean observes that Basic Security Theorem alone is a necessary but insufficient property for a system to be secure. As an example, he presents a variation of BLP with an inverted version of the $\star$-property called †-property for which BST holds but that is not secure in any

real sense. Furthermore, in [35] and [36] he presents another variation of BLP called *System Z* that always downgrades all subjects and objects to the lowest possible level and enters all access permissions in all access matrix cells. Strictly interpreted, System Z is BLP-compliant even if it is obviously contrary to BLP design guidance. Bell presented his countercriticism in [6]. Most notably System Z violates the *tranquility property* principle of BLP which says that the security labels of subjects and objects should not change during system operation.

2.1.2 Biba

The security model developed by Ken Biba [9] was the first model to address integrity in computer system security. In computer security literature, several meanings are associated with the concept of integrity [33]:

- preventing unauthorized users from making modifications
- maintaining internal and external consistency
- preventing authorized users from making improper modifications

The Biba model addresses the most common of these, i.e. its goal is to prevent unauthorized users from making modifications.

The model structure very closely resembles the Bell-LaPadula model and shares its basic concepts such as subjects, objects, accesses, and system security as security-preserving state transitions. It is also a mandatory security model. As previously described, in the BLP model the security labels consist of a sensitivity level and a set of categories. The category concept is adopted as such in the Biba model. However, while in the BLP model the sensitivity levels measure confidentiality, in the Biba model they designate levels of integrity. Integrity is interpreted as the measure of trustworthiness. While BLP sensitivity levels have real-life counterparts in military and government environments, Biba integrity levels don't have such counterparts and consequently it is difficult to assign integrity levels in practice.

The original work suggests five different ways, or policies, to achieve integrity: Low-Water Mark Policy for Subjects, Low-Water Mark Policy for Objects, Low-Water Mark Integrity Audit Policy, Ring Policy, and Strict Integrity Policy. The last one, Strict Integrity Policy, is the best known and in fact the term "Biba model" is customarily used to refer to this specific policy.

The mathematical formulation of the Biba model is analogous to the BLP model. The system consists of:

- A set of subjects S.
- A set of objects O.
- A set of access operations $A = \{read, write\}$.
- A lattice of integrity labels $(L, \geq)$.
- A function $i : S \cup O \rightarrow L$ that gives the integrity label associated with a subject or object.
- A relation $R \subseteq S \times O : s\, R\, o \Leftrightarrow s$ can read o. Defines which subjects are allowed to read objects.
- A relation $W \subseteq S \times O : s\, W\, o \Leftrightarrow s$ can write o. Defines which subjects are allowed to write to objects.
- A relation $I \subseteq S \times S : s_1\, I\, s_2 \Leftrightarrow s_1$ can invoke s_2. Defines which subjects are allowed to invoke actions on other subjects.

The security properties of the Strict Integrity Policy are:

- *Simple Integrity property* (also known as *No Read Down*):

$$s\, R\, o \Rightarrow i(o) \geq i(s)$$

  In other words, a subject may not read information that is less trustworthy than the subject itself. The motivation for this property is to prevent trusted information held by a subject from being contaminated by other, less trusted information.

- *Integrity $\star$-property* (also known as No Write Up):

$$s\, W\, o \Rightarrow i(s) \geq i(o)$$

  In other words, subjects may not modify information that is more highly trusted than themselves. The purpose of this property is to prevent unauthorized subjects from directly modifying trusted information.

- *Invocation property*:

$$s_1 \, I \, s_2 \Rightarrow i(s_1) \geq i(s_2)$$

Invocation is the event where one subject requests another one to perform an action. Since invocation inherently transmits information from one subject to another, invocation is a special case of the write operation. Hence invocation is constrained by a restriction analogous to the integrity $\star$-property.

2.1.3 Harrison-Ruzzo-Ullman

The BLP and Biba models assume that the subjects, objects, and access rights of a system remain fixed. This assumption is not very realistic, and the HRU model by Harrison, Ruzzo, and Ullman [24] specifically addresses this issue. The model was developed in order to investigate safety in an environment where subjects, objects, and access rights change.

Model

The HRU model defines an authorization system that is composed of the following elements:

- A fixed set of access rights R.
- A set of subjects S.
- A set of objects O.
- An access matrix $M = (M_{s,o} \subseteq R), s \in S, o \in O$. The element $M_{s,o}$ records the access rights the subject s has on the object o.
- A set of commands C.

The commands are always of the following form:

if $r_1 \in M_{s_1,o_1}$ **and**
$r_2 \in M_{s_2,o_2}$ **and**
...
$r_m \in M_{s_m,o_m}$
then
op_1
op_2
...
op_n
end

In the command, $s_i \in S, o_i \in O, r_i \in R$, and each op_i is one of the following six primitive operations:

- **enter** r **into** $M_{s,o}$
- **delete** r **from** $M_{s,o}$
- **create subject** s
- **delete subject** s
- **create object** o
- **delete object** o

A system configuration is captured by the triple (S, O, M), where the elements specify the current set of subjects and objects and the current rights in the access matrix. Execution of commands results in a change in system configuration.

Safety Problem

In their work, the authors consider the problem of deciding if in a given system a particular access right may be acquired by an unauthorized subject. This problem is called the *safety* problem for protection systems. A system with the initial state (S_0, O_0, M_0) is said to be safe with respect to access right r if there does not exist a sequence of commands that would eventually enter r in M in a position that originally did not contain r.

The authors prove that the problem is undecidable in its general form, that is there does not exist an algorithm that can decide the safety of an arbitrary authorization system. However, they also note that more restricted

cases of the problem exist that are decidable. As an example they consider *mono-operational systems*, i.e. authorization systems whose commands consist of only one primitive operation. A proof is presented that for mono-operational systems the safety problem is decidable.

2.1.4 Clark-Wilson

David Clark and David Wilson were the first ones to emphasize the importance of integrity in computer security systems. In their noteworthy paper they argued that the contemporary security models originating from military and government environments were not sufficient in the commercial world [11]. The main point is that while the military models are mostly concerned by confidentiality, the most important aspect of commercial security is integrity. This lead the authors to develop another model centered around the concept of integrity.

Integrity Policy

Clark and Wilson observe that in commercial sectors, and in particular in bookkeeping and accounting activities, the major threats to integrity are errors and fraud. They proceed to define a commercial security policy for integrity based on two well-established existing commercial integrity preserving mechanisms: well-formed transactions and separation of duty.

The well-formed transaction principle states that users may not modify data in arbitrary ways, but exclusively by using certified procedures that are known to maintain data integrity. Examples of the well-formed transaction principle include keeping logs to maintain audit trails and double entry bookkeeping.

Separation of duty is mostly concerned with preventing authorized users from making improper modifications such as frauds and thereby violating external consistency.

Model

In the Clark-Wilson model a system consists of the following entities:

- *Constrained Data Items* (CDI)
- *Unconstrained Data Items* (UDI)
- *Integrity Verifying Procedures* (IVP)

- *Transformation Procedures* (TP)

All data in the system is divided into two categories: constrained and unconstrained. CDI contains data that through verification is known to be valid and consistent. All other data belongs to the UDI category. IVPs are used to verify the conformance of all CDI data to integrity specification at the time of invocation. TPs are the essence of well-formed transactions in the Clark-Wilson model. They are trusted procedures that modify CDI data but preserve their integrity. In the Clark-Wilson model, modifications of CDI data are only allowed via TPs. Consequently, if it is assumed that the system starts from a secure state, the system will always be secure with respect to integrity policy since all potential future states can be reached only by transaction procedures that preserve integrity.

The model presents nine rules that must be satisfied in order to comply with the integrity policy. There are two kinds of rules: certification rules (C) that must be satisfied by administrative personnel (security officer, system owner, and system custodian) with respect to the integrity policy and enforcement rules (E) that must be satisfied by the system. The rules are defined as follows:

- C1: (Certification) All IVPs must properly ensure that all CDIs are in a valid state at the time the IVP is run.
- C2: All TPs must be certified to be valid. That is, they must take a CDI to a valid final state, given that it is in a valid state to begin with. For each TP, and each set of CDIs that it may manipulate, the security officer must specify a relation or function that defines that execution. A relation is thus of the form $(TP_i, (CDI_a, CDI_b, CDI_c, ...))$, where the list of CDIs defines a particular set of arguments for which TP_i has been certified.
- E1: (Enforcement) The system must maintain the list of relations specified in rule C2, and must ensure that the only manipulation of any CDI is by a TP, where the TP is operating on the CDI as specified in some relation.
- E2: The system must maintain a list of relations of the form: $(UserID, TP_i, (CDI_a, CDI_b, CDI_c, ...))$, which relates a user, a TP, and the data objects that TP_i may reference on behalf of that user. It must ensure that only executions described in one of the relations are performed.

- C3: The list of relation in E2 must be certified to meet the separation of duty requirement.
- E3: The system must authenticate the identity of each user attempting to execute a TP.
- C4: All TPs must be certified to write to an append-only CDI (the log) all information necessary to permit the nature of the operation to be reconstructed.
- C5: Any TP that takes a UDI as an input value must be certified to perform only valid transformations, or else no transformations, for any possible value of the UDI. The transformation should take the input from a UDI to a CDI, or the UDI is rejected. Typically, this is an edit program.
- E4: Only the agent permitted to certify entities may change the list of such entities associated with other entities: specifically, the entities associated with a TP.

The first three rules (C1, C2, E1) guarantee internal consistency in the system. The following two rules (E2, C3) handle external consistency via separation of duty and with the help of the E3 rule that governs user authentication. C4 says that an audit trail has to be maintained by requiring TPs to log all relevant information. C5 defines how unconstrained input is imported to the system. Finally, E4 enforces the separation of duty principle upon the certification operations.

2.1.5 Brewer-Nash

In their 1989 paper [10] Brewer and Nash published a security model based on a well known commercial security policy called Chinese Wall. This policy originates from the code of practice observed by analysts providing corporate business services. A single analyst may deal with information concerning several different client companies. It is assumed that when consulting a client company an analyst obtains inside information regarding it. Naturally, the inside information entrusted to the analyst must not be used to benefit any of the client's competitors. Thus the gist of the policy is that an analyst may not consult a client company if he has already obtained information regarding another company in the same competitive sector.

Consequently, an analyst is free to choose to access information about any company in a new competitive sector. But once he has obtained information about a company in that sector, he cannot access information regarding any other company in that sector anymore.

Model

The Brewer-Nash model is constructed in a very BLP-like fashion so that it can be easily compared to the BLP model. It adopts the BLP concepts of subjects, objects, and security labels. Subjects represent users and programs that act on their behalf. Objects are pieces of information concerning a single company. They are arranged in a three-level hierarchy as illustrated in Figure 2.3. At the lowest level there are information *objects* which are associated with exactly one company. At the intermediate level there are *company datasets* each of which groups together all objects concerning that company. Company datasets are identified by the name of the respective company. At the highest level there are *conflict of interest classes* which consist of all datasets concerning companies in competition with each other. Conflict of interest classes are named after their business sectors. Finally, a boolean matrix N records the access history.

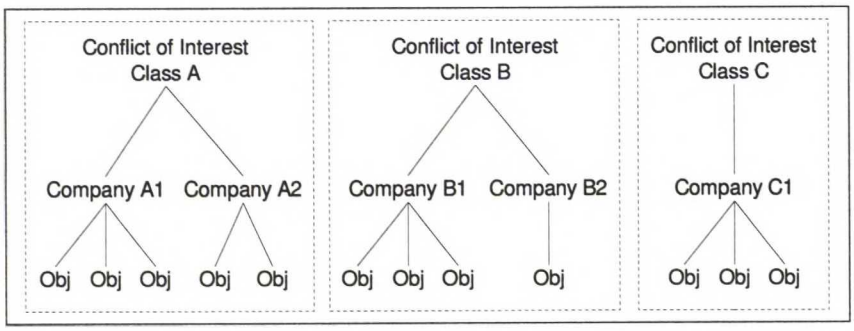

Figure 2.3: Brewer-Nash object hierarchy

Brewer and Nash make a distinction between unsanitized and sanitized information objects. Unsanitized objects are company specific information and hence subject to protection measures. Sanitized objects are public information not related to any specific company. They are freely accessable and not subject to controls by the model.

Formally, the model can be defined as follows:

- A set of subjects S.

- A set of objects O. Each object $o \in O$ is associated with a security label $(x(o), y(o))$. $y(o)$ denotes the company to which o belongs and $x(o)$ denotes the conflict of interest class to which the company belongs. For sanitized objects, $x(o) = y(o) =$ PUBLIC.
- A boolean matrix $N = (N_{so}), s \in S, o \in O$. N_{so} is true if and only if subject s has previously accessed object o.

The Chinese Wall policy is enforced by two mandatory rules:

- *Simple Security Rule*: Subject s can read object o if and only if

$$\forall \phi \in O, N_{s\phi} = true : (x(\phi) \neq x(o)) \vee (y(\phi) = y(o)).$$

  In other words, the subject may read the object if it belongs to a company dataset that the subject has already accessed, or if it belongs to a conflict of interest class that the subject has never accessed before.

- $\star$*-Property Rule*: Subject s can write object o if and only if

$$\begin{aligned} &N_{so} = true \wedge \\ &\nexists \phi \in O, N_{s\phi} = true : \\ &\qquad (s \text{ can read } \phi) \wedge (y(\phi) \neq y(o)) \wedge (y(\phi) \neq \text{PUBLIC}) \end{aligned}$$

  In other words, the subject may write to the object if 1) the subject can read the object and 2) the subject cannot read any object which is in a different company dataset to the one for which write access is requested.

The simple security rule captures the Chinese Wall policy with respect to read access. However, the $\star$-property rule is still needed to prevent indirect information leaks because information may still indirectly leak between competitors if two analysts consulting two competitors share a common client in another conflict of interest class. A malicious program such as a Trojan Horse could write information about one competitor to the common client in the other conflict of interest class and that information could then be read by the other analyst.

As an example of an indirect leak consider two oil companies OC_1 and OC_2 and one bank B. Analyst A_1 consults B and OC_1 while another analyst A_2 consults B and OC_2. Because of the simple security rule, A_1 cannot

directly read OC_2 information and A_2 cannot read OC_1 information. However, a Trojan Horse used by A_1 could write OC_1 information to B and that information could later be obtained by A_2.

The underlying motivation for the $\star$-property is very similar both in BLP and Brewer-Nash models. The basic confidentiality policy is enforced by the simple security rule. In both models it relies on the assumption that security labels of objects correctly indicate their level of confidentiality (classification and categories in BLP, company name in Brewer-Nash). If write operations were allowed without any restrictions it would be easy for a malicious program to violate that assumption (e.g. in BLP by writing secret information to unclassified objects and in Brewer-Nash by writing company A information to company B objects.)

As observed by Sandhu [58] the Brewer-Nash $\star$-property implies the following:

- A subject that has read objects from exactly one company dataset can write to that dataset.
- A subject that has read objects from two or more company datasets cannot write at all.

So, the $\star$-property has the undesirable side effect of preventing consultants from having more than one client company. It is true that if a computer program executing on behalf of a consultant holds information about more than one company an indirect leak will inevitably be possible. However, the $\star$-property restrictions should really be imposed on computer programs rather than on the consultants since consultants can leak information outside of the computer system in any case. The undesirable side effects can be avoided by executing programs with restricted permissions so that information concerning at most one company can be read at a time. So in effect a program is executed in a fresh new session that does not record information from any previous session and whose set of "already accessed companies" contains at most one entry.

In [58] Sandhu also demonstrates that contrary to authors' belief, the Brewer-Nash model (without the aforementioned side effects) can be represented as a lattice based BLP variant. In his work Sandhu generalizes the model further so that objects may contain information pertaining to several companies of different conflict of interest classes.

2.1.6 Role Based Access Control

Recent research on access control models has concentrated around a model called Role Based Access Control (RBAC). The first steps of RBAC development were taken in the early 1990s when it was understood that the contemporary MAC and DAC models were not suitable for modeling security policies of databases and commercial organizations in civilian environments [3, 51, 21]. The central idea of RBAC is to define access permissions in terms of roles rather than individual users. In large organizations there are possibly tens of thousands of users which makes it infeasible to administer permissions for each individual user. Furthermore, permissions are constantly changing since employees come and go and people change job titles and responsibilities. Administration is facilitated considerably if permissions are assigned to a small number of relatively static roles that represent natural and business-like organizational concepts such as job titles or responsibilities. Individual users can then be assigned to roles independently of permissions and obtain permissions through their roles.

Essentially, RBAC is not a complete security model as the models presented above are. It leaves considerable degrees of freedom for system implementors and administrators in adapting the model to a given organization's needs. In particular, it does not mandate any security policy. Rather RBAC is a useful modeling technique for constructing more refined models that fulfill security policies. Actually it has been shown that the RBAC model is flexible enough to express several well-known policies implemented by traditional models such as MAC, DAC, and Clark-Wilson [48] [44].

While research on RBAC is going on there have been efforts to standardize some of most common and useful RBAC concepts [56, 57]. The following sections present on overview of the NIST RBAC models.

NIST RBAC Model

Several new ideas and features have been proposed around the core RBAC concept. The NIST model presents the most essential of these features in a stepwise manner using four reference models of cumulatively increasing features: Flat RBAC, Hierarchical RBAC, Constrained RBAC, and Symmetric RBAC.

Flat RBAC The flat model represents the minimum set of core features that any implementation should support in order to be considered RBAC compliant. The flat RBAC model is illustrated in Figure 2.4. It consists of the following entities:

- U: a set of users. Users are human beings or agents running on their behalf.
- R: set of roles. Roles represent job titles and responsibilities within an organization.
- P: set of permissions. Permissions are positive authorizations to perform an action on an object. Permissions can represent actions on any level of abstraction such as low-level filesystem permissions or high-level business transactions; their semantics is not to be interpreted by the RBAC system.
- $UA \subseteq U \times R$: a relation that defines the assignment of users to roles.
- $PA \subseteq R \times P$: a relation that defines the assignment of permissions to roles.

The basic features of the flat RBAC model are:

- Users obtain permissions through membership in roles.
- User-role assignment (UA) is many-to-many; i.e. a user can be a member of several different roles, and a role can have several different members.
- Permission-role assignment (PA) is many-to-many; i.e. a permission can be assigned to several different roles, and a role can hold several different permissions.
- User-Role assignment can be reviewed efficiently. This means that the UA data must be stored and handled in a way that allows an efficient computation of the following two functions: $roles : U \rightarrow 2^R$ (to find the roles a user is member of) and $members : R \rightarrow 2^U$ (to find the members of a role).
- A user can have many roles active simultaneously rather than only one at a time. Users sign in the system via sessions in which they may activate only a subset of their available member roles to keep

their permissions to minimum according to the principle of least privilege. However, the session concept is not explicitly defined in the flat model.

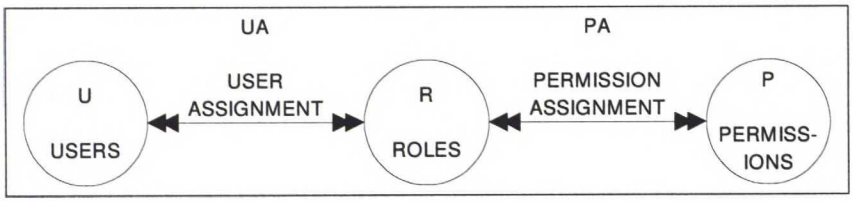

Figure 2.4: Flat RBAC

Hierarchical RBAC The Hierarchical RBAC model (illustrated in Figure 2.5) enhances the flat model by adding *role hierarchies*. In mathematical terms, a role hierarchy in its most generic form is an arbitrary partial ordering on the set of roles. They are a natural way of modeling organizational structure in terms of senior and junior roles. A sample role hierarchy is illustrated in Figure 2.6.

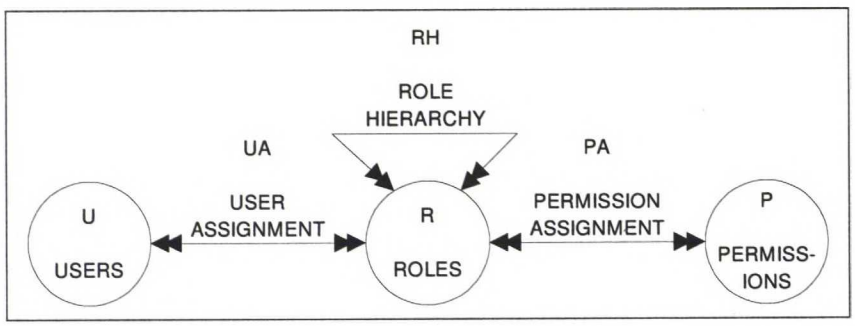

Figure 2.5: Hierarchical RBAC

The basic interpretation of role hierarchies is that senior roles automatically inherit all permissions from their junior roles. This type of hierarchy is called *permission inheritance hierarchy*. Role inheritance makes security administration easier because permissions of entire role hierarchies can be assigned to a user by a single assignment to a senior role. Without inheritance hierarchies a user should be assigned to each of a potentially large number of roles, or permissions should be redundantly duplicated in several roles.

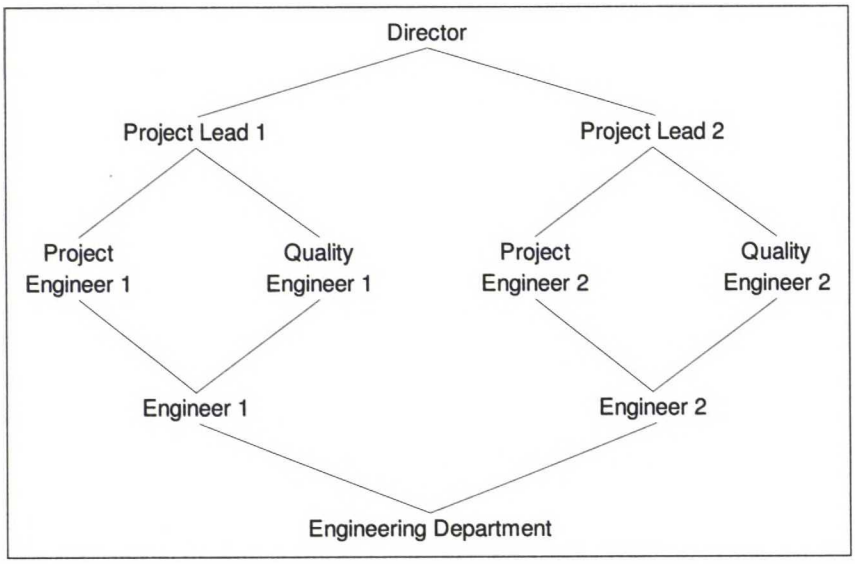

Figure 2.6: Sample role hierarchy

However, it is not always desirable for senior roles to inherit all junior role permissions. For example a senior user may want to deliberately reduce their permissions when performing tasks that don't require them, following the principle of least privilege, e.g. to limit the effects of a trojan horse. Or in other cases the inherited roles may be in conflict with separation of duty constraints. In such cases the role hierarchy is interpreted so that senior roles do not automatically inherit all junior roles but instead may choose at will which ones to activate. This interpretation is called *role activation hierarchy*. In some cases such as with dynamic separation of duty it is possible that the permission inheritance and role activation hierarchies are distinct, the latter being a superset of the former [54].

Constrained RBAC Augmenting the hierarchical model with constraints leads to the Constrained RBAC model. Constraints enable security administrators to enforce separation of duty (SOD) policies such as preventing a user from acting in both the roles of purchasing officer and accounts payable officer.

Separation of duty constraints can be either static (SSD) or dynamic (DSD). Static constraints impose restrictions on the assignment or users to roles as mutual exclusivity requirements. Any user may be assigned to at most one role in a set of roles marked as mutually exclusive. For example,

a static separation of duty policy could specify that the aforementioned purchasing officer and accounts payable officer roles are mutually exclusive. Like permissions, also static constraints are inherited through the role hierarchies. So, if a role is mutually exclusive with another role, all its senior roles will automatically inherit the constraint and thus be mutually exclusive with the latter role, too.

Dynamic constraints restrict the roles a user may have active during a session. This is useful when a conflict of interest situation does not follow immediately from static role membership but occurs only if these roles are active simultaneously during the same session. In general, constraints in terms of active roles allows for more fine-grained constraints than static constraints alone. Dynamic constraints are not inherited within a role hierarchy. Figure 2.7 is an illustration of the Constrained RBAC model with dynamic separation of duty which is identical to the static version apart from the addition of the constraints arrow leading to session objects.

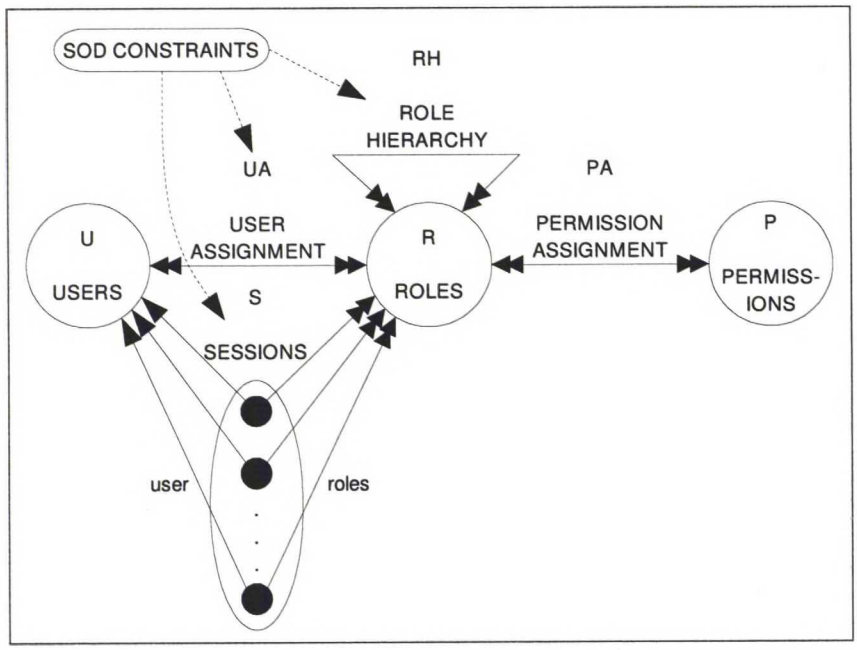

Figure 2.7: Constrained RBAC with dynamic separation of duty

Symmetric RBAC Organizations are constantly in change because new employees are hired and old ones change titles, responsibilities and even retire. In administering security of an organization all users should obtain

all the privileges they need to perform their due duties but no other privileges. In these dynamic conditions administration is a challenging task. The problem is complicated even further if the system is distributed in the sense that it covers several distinct but collaborating organizations.

One useful tool that aids in security administration is the user-role review introduced in the flat RBAC model. The symmetric model builds on the constrained model, adding permission-role review as a new requirement. Given a user or a role, administrators can ask for its set of accessible objects or set of available permissions as (operation, object) pairs. The reviewer can choose to list only the directly assigned objects or permissions, or to include all inherited ones as well. And finally, in distributed systems the review must provide for selecting the target systems for which the review should be performed.

RBAC Administration

As noted before, RBAC is very flexible in terms of what kinds of security policies it can express and does not enforce any specific policy itself. Instead security administrators are provided a lot of freedom to customize the model to suit their organizational security policies. Administering large systems is a very challenging task. System administrators are faced with the problem of policy refinement as discussed in Section 1.4.2. In the early days of RBAC little attention was paid to administration and a centralized administration model was assumed. Later it became clear that large systems cannot be effectively managed by a central authority but instead administrative responsibility must be decentralized, distributed and delegated to smaller units in the organizational hierarchy.

So far research in RBAC has not yet produced proven best practices or off-the-shelf recipes that would specify how to configure RBAC to meet the most common security policy objectives. The connection between organizational structure and responsibilities and the RBAC components has been studied [50, 46]. Using RBAC in inter-organization collaboration has also been studied [26, 25]. Some basic mechanisms to support decentralized RBAC administration have been published as the Administrative RBAC 97 (ARBAC97) model [55] and its improved version ARBAC02 [47]. The following section provides a brief overview of the ARBAC02 model.

Administrative RBAC model The ARBAC model addresses the problem of decentralized security administration in the RBAC model. It specifies

constraints that can be used to delegate administration authority in a controlled manner. Figure 2.8 illustrates the ARBAC02 model components.

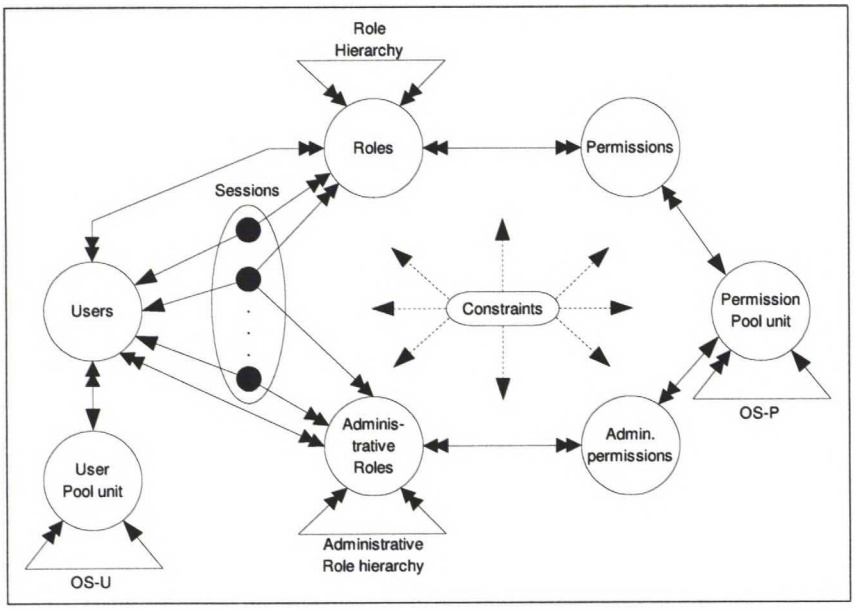

Figure 2.8: ARBAC02 components

Administrative roles and permissions were already present in the RBAC96 model [56]. The underlying idea is to separate administrative tasks from normal tasks and then to use existing RBAC principles to control itself. Administrative roles and permissions are disjoint from normal roles and permissions, respectively.

For administrative purposes, the organization's structure is explicitly modeled. More precisely, it is the duty of the Human Resources (HR) department to maintain a tree structure of the organizational units and assign users to these units. This tree structure is called the user pool. Likewise, it is the duty of the Information Technology (IT) department to maintain a (inverted) tree structure of organizational units and to assign resource permissions to its nodes. This structure is called the permission pool. Finally, security administrators assign users from the user pool and permissions from the permission pool to actual roles. This role administration concept is illustrated in Figure 2.9.

The administrative operations in ARBAC are:

- Assigning and revoking users from roles (URA)
- Assigning and revoking permissions from roles (PRA)

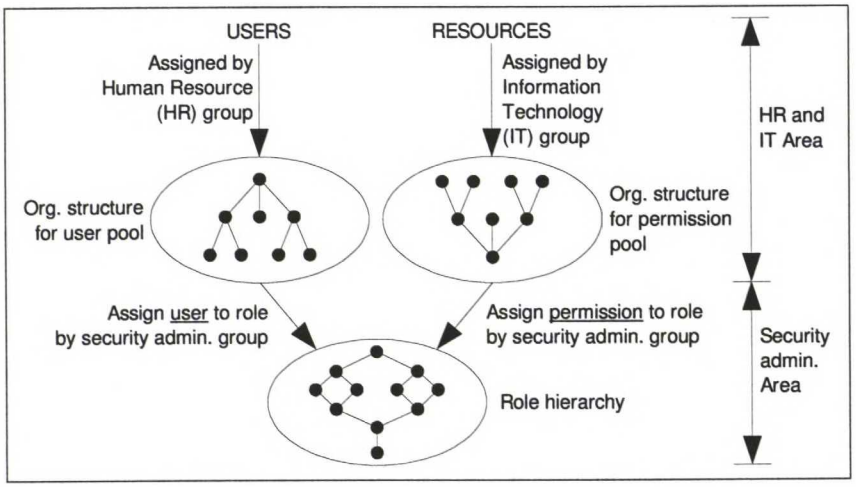

Figure 2.9: Role Administration Concept in ARBAC02

- Modifying the role inheritance hierarchy (RRA)

Authorized administrative operations are captured by five relations: *can-assign, can-revoke, can-assignp, can-revokep,* and *can-modify*, respectively. For example, user to role assignment is governed by the *can-assign*(x, y, z) relation. x denotes the administrative role whom this rule concerns. y is an arbitrary boolean expression called *prerequisite condition* whose terms are either role names or organizational units from the user pool. The terms are true if the user in question belongs to the role or unit in question. Terms that denote organizational rules are prefixed with an at sign ('@') and terms that denote roles are written as such. z is a *role range* that denotes a set of roles between two endpoint roles in the role inheritance hierarchy.

The meaning of the *can-assign* relation is that an administrator that is a member of administrative role x or any of its superior roles may assign users to the roles specified by z if the prerequisite condition holds for the user in question. For example, *can-assign(Engineering Dept, @Engineering* $\wedge$ $\overline{@Sales}$ $\wedge$ *proj Y, [proj X lead, proj X engineer])* means that engineering department security officers may assign users to any roles between the project X lead and engineer roles in the inheritance hierarchy if the users belong to the engineering unit but not to the sales unit and if they are not already members of the project Y role.

Revocation of users is controlled by a similar *can-revoke*(x, z) relation. The difference is that there are no prerequisite conditions but instead ad-

ministrators can revoke any user from the given role range. ARBAC02 differentiates between *weak revocations* and *strong revocations*. A weak revocation removes a user only from the role to which it explicitly belongs. Strong revocations on the other hand revoke a user also from all the roles that are senior to the one being revoked.

The permission controlling relations *can-assignp*(x, y, z) and *can-revokep*(x, z) are analogous to the *can-assign* and *can-revoke* relations discussed above.

To make decentralization of administration possible the ARBAC model allows also the authority to manage a subset of the role inheritance hierarchy to be delegated. The operations for managing the role hierarchy are create role, delete role, create inheritance edge, and delete inheritance edge. The *can-modify*(x, z) relation specifies that an administration that is a (possibly indirect) member of role x can perform the operations in the subset of role inheritance hierarchy specified by the role range z. Furthermore, RRA distinguishes between three types of roles: *abilities, groups,* and *UP-roles*. Abilities can contain only permissions, groups can contain only users and UP-roles can contain both. The model still adds analogous relations for controlling management of these different types of roles: *can-assigna, can-revokea, can-assigng, can-revokeg*.

ARBAC specifies how organizations' chief security officers may delegate administrative permissions to subordinates. The administrative operations of ARBAC are assignment and revocation of users to roles (URA), permissions to roles (PRA), and roles to roles (RRA).

2.2 Recent pragmatic models

2.2.1 Java security

Overview

Secure programming has been one of Java's goals right from the beginning. The Java development environment includes lots of different features that aid in producing secure programs. There are features built in the language itself such as strong typing, bounds checking, automatic memory management, absence of pointers, and bytecode verification. In addition, there are several APIs for taking advantage of existing security technologies such as Java Cryptography Extension (JCE), Java SecureSocket Extension (JSSE), Java Authentication and Authorization Service (JAAS), Java Certification

Path API, and Java GSS-API.

However, this work does not consider Java language features or security APIs. Instead, it addresses the underlying security architecture which is concerned with controlling the authorized execution of protected operations by Java code.

Since the initial Java release many security related features have been improved and new ones have been added. The first versions were mainly concerned about threats by malicious code such originating from untrusted hostile networks. Remote code was always run in a restricted runtime environment called *sandbox*. Later it became possible for cryptographically signed remote code to run with full privileges as any other trusted code. In the most recent model code is not simplistically divided to trusted and untrusted code but instead different pieces of code can be assigned different privileges in a fine-grained manner. The presentation below is based on Java 2 SDK Standard Edition version 1.4 [23, 28, 64]. This version provides an extensible policy-based authorization model that can be used both by standard Java libraries and application code.

Protection domains

In Java, every frame in the call stack of a thread of execution has an associated *protection domain*. A protection domain contains the following information:

- An URL that specifies where the class that implements the corresponding method originates from
- Any cryptographic certificates that were used to verify the signatures of the class
- Properties of the subject (user or service) on behalf of which the code is executing
- A set of permissions

The subject properties include identities and credentials. Identities are represented by instances of *principal* classes that are basically intended to denote identities such as user account names or X.500 identities. However, the fact that custom principal classes can be created allows them to be used for describing group or role memberships or any other arbitrary subject properties as well [28].

Protection domains contain a set of static permissions that are fixed when the corresponding class is loaded and its protection domain is created. Furthermore, when a permission in a protection domain is checked the *current policy object* (see below) is queried for additional dynamic permissions that may change at runtime.

Permissions and policies

Protected operations are represented by *permission* classes. Permissions have a mandatory target name part and an optional set of actions. For example, the name of a file permission is a file system path and its actions include `read` and `write`. The name part of a network socket permission is a host-port specification and its actions include `accept`, `connect`, and `listen`. Some permissions such as runtime permissions only have a target name but no actions, e.g. `setSecurityManager`. Developers can freely create new permissions to represent application specific protected operations such as business transactions.

In the Java security architecture a policy is a mapping from the characteristics of a protection domain to a set of permissions. This mapping is carried out by a system-wide runtime security policy object. Custom policy objects can be created in order to implement a variety of different policies. Furthermore, there exists a policy reference implementation that uses a simple text-based configuration file to specify policies. The policy configuration file is basically a list of `grant` statements that allow certain permissions if the protection domains in question meet the given conditions. The types of conditions supported are:

- `signedby <signer name>`: the code was signed by the given signer
- `codebase <URL>`: the java class was loaded from the given URL
- `principal <class> <name>`: the subject has a principal of the given class with the given name

Figure 2.10 illustrates a simple example policy configuration file.

Authentication

Since Java 2 SDK version 1.4 authentication and authorization are taken care of by the Java Authentication and Authorization Service (JAAS). It provides a framework resembling Pluggable Authentication Modules (PAM)

```
// code originating from company intranet running
// as the admin role can modify password data
grant codebase "http://intra.acme.com",
      principal com.acme.Role "admin" {
   permission java.io.FilePermission "/etc/passwd", "read,write";
}
```

Figure 2.10: Example Java policy configuration file

[53] that takes care of authentication procedures on behalf of the application. Authentication modules and configuration can be modified by administrators without changing the application itself.

As a result of a successful authentication process the application is provided a `Subject` class instance that contains properties of the subject that were collected by the authentication modules during the process. The properties may include identities, credentials, and also any other arbitrary custom properties such as role membership if supported by the application and the authentication modules.

Subsequently, if the application so wishes it can ask the runtime environment to continue execution in the role of the authenticated subject. This is accomplished with one of the `doAs()` -family methods of the `Subject` class. From that moment on, all authorization checks for the current method and any other methods below it in the call stack are performed with respect to the newly assumed subject role.

This feature is useful, for example, in situations in which a server application is willing to act on behalf of a user and assumes the user role so that any subsequent authorization will based on that user's permissions.

Authorization

In Java, access control checks are performed at runtime by a system `AccessController` class. Any code, be it system or application, willing to check that the current *access control context* be allowed to execute a protected operation can call its `checkPermission` method, passing it a `Permission` object presenting the permission in question. The access controller then consults the current security policy and compares it to the current context. If the permission is granted execution continues normally. Otherwise a security exception is raised.

The current access control context essentially consists of the protection domains of the stack frames of the current thread of execution. Because code originating from different sources such as system libraries, locally installed application code, and remote application code can arbitrarily call methods on each other, the protection domains in a stack are likely to contain different permissions. Now, authorization decisions can not be based solely on the permissions of the protection domain of the deepest stack frame. Doing so would allow less privileged code to obtain more privileges by calling more privileged code (e.g. when application code calls system code). Instead, it is necessary that the permission be present in all the protection domains of the call stack. Thus the effective set of permissions is the intersection of all permissions of the protection domains in the call stack.

However, this intersecting method poses another problem. Namely, it prevents privileged code from performing protected operations when called by unprivileged code. In cases in which code having the correct permissions wants to perform the operation independently on by whom it is being called it can mark itself as "privileged" code using the `doPrivileged()` family of methods of the `AccessController` class. Once called, any authorization controls on and below this stack frame ignore the preceding call history.

To summarize, a method can perform a protected operation if the corresponding permission is available both to the method itself and to every other method in the current call stack, up until to the last method marked as privileged.

Access control contexts

Normally permissions are checked with respect to the *current* access control context. However, sometimes it is useful to check against a *different* context. One example case is when another thread is performing the check on behalf of another one. Another example is when a server creates an access control context after authenticating a user, but not all the sensitive work is initiated by the same method.

For these purposes, a snapshot of the current access control context can be obtained and saved in an `AccessControlContext` object. Later on, permissions can be checked with respect to the saved context instead of the current context at the checking time.

2.2.2 Firewalls

Firewalls have become an obligatory part of any organization's network security arsenal. In theory all systems inside the network could protect themselves and no central measures would not be needed. However, in practice all systems have vulnerabilities. This makes firewalls an economic solution because they allow for a centralized way of access control that is relatively easy to manage. While not an ideal situation, this can help to enable vulnerable systems to continue operation inside the protected network. Otherwise those systems would have to be taken down or to be continually updated with latest security patches, which currently takes a lot of administrative effort. Moreover, often patches arrive too late or are not available at all, as is the case with legacy systems.

Firewalls only solve a small part of network security problems. The most often cited problem is that of insider threats. As firewalls are usually deployed at the network perimeter, they cannot protect from any flawed or malicious systems or users inside the network. This problem is exacerbated by the increasing use of wireless networks and mobile code. Another problem is that firewalls only have access to limited information which is not always sufficient to make good enough access control decisions. Finally, firewalls may form a network performance bottleneck themselves since they have to control all the traffic entering and leaving the organization's network.

Model

In the literature there have been some efforts to model firewall concepts [60, 5]. The following sections present an overview of firewalls at a high level of abstraction.

The fundamental role of a firewall is to protect a set of trusted systems from a set of untrusted systems by separating the two with a controlled barrier. The most common case is that of separating a trusted organization's internal network from the untrusted public Internet. Another, increasingly common case is protecting private users from the public Internet by software firewalls installed on personal home computers. In a more general sense, there could be any number of hosts and networks separated by a firewall an arranged in various subnet and demilitarized zone (DMZ) configurations.

The basic functions of a firewall are access control and auditing. The

access control function decides which network packets are allowed to traverse and which ones should be denied. The audit function allows administrators to monitor the network by logging information concerning rejected or otherwise suspicious packets. These basic concepts are illustrated by Figure 2.11.

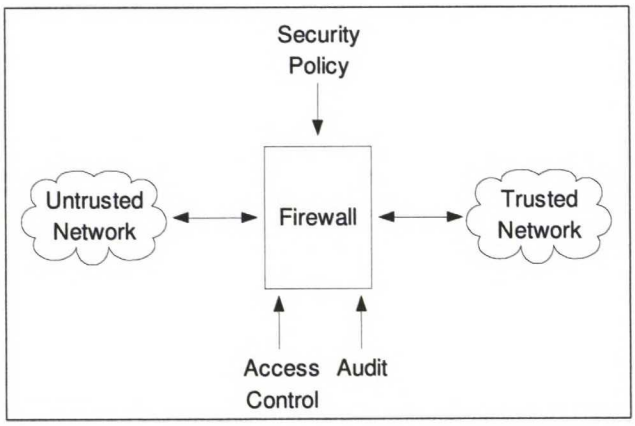

Figure 2.11: Simplified firewall concept model

In addition to these basic functions, modern firewalls have lots of other features and properties as well. The following is a list of (not fully orthogonal) characteristics based on which the existing firewall space may be dissected.

- *Position in the network layer stack.* Most firewalls are capable of inspecting packets at network (IP) and transport (TCP, UDP) levels which allows for basic access control based on network topology and the set of services. Some firewalls additionally understand application level data (such as SMTP and HTTP) which allows for more fine-grained and sophisticated access control.

- *Statefulness.* The simpler firewalls do not store any data on traffic that they have analyzed in the past. Hence they must completely base their access control decisions solely on the current packet at hand. On the other hand, more advanced firewalls maintain some state information. This allows them to e.g. track TCP connections or HTTP session states and in general to make more sophisticated decisions.

- *User transparency.* Filter-type firewalls operate in a transparent fashion that is invisible to users (as long as access is allowed). On the

other hand, some proxy-type firewalls require configuration at the users' systems such as setting a proxy server address. Even though proxy-type firewalls usually operate at the application layer, not all application layer firewalls necessarily require user configuration.

- *Address translation.* Generally it is considered good practice not to reveal any information concerning internal network topology to outsiders. Therefore many firewalls hide that information by dynamically translating internal network addresses to a predefined address space (which is most often the address of the firewall itself). This technique is called Network Address Translation (NAT). It may also be used to conserve public IP address space by utilizing unroutable private IP addresses in the internal network.

- *Centralized vs. distributed.* The traditional firewall as a centralized choke point between external and internal networks has been challenged by the problems presented above (insider threats, wireless networks and dial-up access, mobile code, limited available information at the network perimeter, and performance bottleneck). Some problems can be countered by removing the central choke point and instead distributing firewall functionality elsewhere the internal network [8, 27]. Distributed firewalls can also improve resistance to attacks by adding to depth of defense.

- *Hardware vs. software.* Modern operating systems have all the necessary facilities for building a software-based firewall. For performance reasons however, high-end firewalls are directly implemented in hardware.

- *Protection domain.* Traditionally, firewalls are used to protect entire networks. However, software-based firewalls are increasingly being used on personal computers to protect only the computer in question.

Policies

The following sections illustrate the most common high level policy requirements and how they map to low level firewall configuration.

High level objectives The most fundamental factor that affects firewall policies are the services that must be made available from the trusted network to the untrusted network and vice versa. Typically, organizations only

allow outside access to a small set of well-defined internal services and by default deny all other access. Some organizations may only allow access to a predefined set of external services and deny everything else, while others allow all external services by default but deny some specific services.

In addition to the quite static set of allowed services, there are some more dynamic factors that can affect a firewall policy. For example, an organization may decide to allow some services only during certain time ranges such as business hours or holiday seasons. Additionally, an organization may have an intrusion detection system that maintains a "defcon" level, changes in which could involve change in firewall behavior. Other security alerts such as vulnerabilities in software used by the organization or alerts about ongoing Internet worm attacks could also serve as stimulus for dynamically adjusting firewall behavior.

Furthermore, a personal software firewall that protects a mobile device is likely to need frequent configuration updates since the network connection, topology, and the set of available services changes more often than in organizations with static networks and services. If the firewall policy itself is not aware of mobility, the firewall configuration may have to be manually changed when the device is taken into a new network environment.

Low level configuration The low level configuration of most modern firewalls is an ordered list of firewall rules which consist of two parts: a packet matching condition and an action. When a network packet arrives at the firewall, the firewall goes through the rules one by one, until it finds a rule that matches the packet. Then the rule's action part defines if the network packet is either allowed to or denied from traversing the firewall, potentially auditing it.

Network packet matching is based on the various data present in packet headers, and to some extent in payloads. TCP/UDP port numbers can be used to differentiate between services and host names and IP addresses can be used to differentiate between users or systems. Naturally, this data is not completely reliable since services may run on non-standard port numbers, host addresses can be misconfigured or spoofed and so on. Furthermore, tying access control decisions to low-level packet header data makes the policy fragile with respect to even small changes in the network topology.

Firewall policy management Without proper tool support, firewall management in large organizations can be a very laborious, difficult, and error-

prone task [32]. In the worst case, an administrator has to manually configure a multitude of diverse devices, each of which has a different low-level configuration language, using unintuitive command line tools. Therefore, modern firewall management tools usually have some more advanced features:

- *Device independence.* A firewall can be configured using a generic language that is not directly tied to any vendor's configuration language. The management tool is able to translate the configuration to the specific devices the organization uses.
- *Centralized administration.* Policies can be specified in a central location and they are automatically distributed to the firewalls in the organization's network. All firewalls can be monitored centrally.
- *Graphical user interfaces.* Instead of having to manually edit text-based configuration files or to learn command-line tools, firewall management is performed using intuitive graphical user interfaces.
- *Policy abstractions.* Instead of specifying configuration with respect to low-level information about host addresses, port numbers, interface names etc., tools can introduce higher level concepts such as trusted and untrusted networks and services, and separate the concepts from their low-level implementation.
- *Policy analysis.* In large networks firewall policies grow complex and errors are easily introduced especially if policies are manually created. Tools may verify the conformance of firewall configuration to a security policy and identify conflicts and potential vulnerabilities in it [32, 68, 20].

Even with modern firewall management tools, policies are still closely tied to the topology of the organization's network. As discussed above, the low-level topology information cannot be relied on, and may not be fine-grained enough for access control decisions. However, distributed firewalls promise improvements since more information is available to end hosts than to the intermediate firewall. For example, end hosts are able to decrypt encrypted connections, differentiate between the operating system processes and users that send or receive traffic, and have better knowledge about application-level packet payload.

It is also recognized that distributed firewalls are fairly easy to circumvent if the firewall runs in the same host operating system [49]. In the DARPA Autonomic Distributed Firewalls project, a firewall has been embedded to a network interface card to produce an Embedded Firewall [49, 38].

2.3 Customizable policy models

The following section presents the Policy Common Information Model, which supports custom security policies. But unlike the models in Section 2.4 it does not provide a language for specifying them. Rather it provides a conceptual model of policies.

2.3.1 Policy Common Information Model

Background

The Common Information Model (CIM) is an ongoing standardization effort to ease the management of large distributed enterprise computing and networking environments. It is being developed by the Distributed Management Task Force (DMTF) and is based on earlier work at DMTF including System Management BIOS (SMBIOS), Desktop Management Interface (DMI), and Directory Enabled Networking (DEN). It defines an open and extensible framework of standard interfaces for producing and consuming management information, simplifying management application development and allowing products from different vendors to interoperate.

The Policy Core Information Model (PCIM) is an extension to CIM. It provides generic object classes for describing policy related information. It is a joint development effort by the Internet Engineering Task Force (IETF) Policy Working Group and DMTF.

Web Based Enterprise Management (WBEM) is a DMTF initiative that specifies how CIM can be utilized using standard Internet technologies XML and HTTP.

All the CIM standards are work in progress. Currently there is ongoing security related effort at least in three working groups: Security Protection and Management Working Group, Policy Working Group, and User and Security Working Group.

Common Information Model

CIM is an abstract, conceptual information model for describing the entities and their relationships in a management environment. It is technology and data format independent. It adopts an object-oriented design strategy to leverage the benefits of object oriented systems such as abstraction, reuse, and extensibility.

CIM Meta Schema is a formal definition of the CIM model. It consists of definitions for elements like classes, properties, methods, and associations. The schema classes are constructed out of these primitive elements.

In CIM terminology, schemas are sets of classes that model some domain of the real world. Schemas always have a single owner and a name. They are used for administration and class naming. All class names within a schema must be unique. Classes are divided to two categories: classes that describe the managed elements and association classes. Association classes represent relationships such as aggregation between objects and are also modeled as classes.

The CIM Schema is divided to three layers: Core Model, Common Model, and Extension Schema. The Core Model is a small, stable set of fundamental classes that are applicable to all management areas. The Common Model consists of basic schemas for various areas of management such as systems, applications, networks and devices. However, the classes of the Common Model are technology and platform independent. The Common Model is currently being expanded to cover new areas. Extension schema classes are used to extend the Common Schema classes with technology or vendor specific additions .

Management information in CIM is defined in Management Object Format (MOF) which is a text based language based on the Interface Definition Language (IDL). The MOF class specifications are also accompanied by a graphical Unified Modeling Language (UML) representation. Appendix B contains the definition of a public key certificate in Managed Object Format.

Policy Common Information Model

Introduction and Overall Concepts The policy related CIM extension models are being developed in parallel by DMTF and IETF and are thus different specifications but nonetheless very similar. The DMTF model is documented in the CIM Core Policy Model [17] and the IETF model in [43]. The discussion below is based on the IETF PCIM and PCIM Extensions

documents. The purpose of PCIM is to enable administrators of complex, multi-vendor enterprise environments to represent and manage policies. The policy framework is built to be capable of representing policies about anything. Domain specific extensions such as QoS or security extensions will be separately developed that will leverage the existing policy framework.

In the PCIM model, policies are basically coherent sets of condition-action rules that are targeted to some roles. Roles represent a grouping of managed elements. Thus for example, instead of writing individual device- or host-specific rules, a rule can be targeted to a group of elements, which allows the policies to scale in large scale environments. Rules can be combined to form hierarchies and these can be reused as parts of other rules. This allows for flexible management and reuse of policies.

Rules and Grouping PCIM policies are represented as sets of policy rules. These individual rules are represented as subclasses of `PolicyRule` and are then combined to form the rule sets. Rule sets are represented as subclasses of the abstract class `PolicySets`. Rule sets may be nested, i.e. may contain other rule sets as subsets. The nesting relationship is indicated using the `PolicySetComponent` aggregation class which collects `PolicySet` instances as parts of the containing `PolicySet`. There are two concrete subclasses of `PolicySet`: `PolicyRule` and `PolicyGroup`. Since `PolicySetComponent` is an association between `PolicySets`, both policy rules and policy groups may contain other subrules and groups as subsets. Containment relations are not allowed to form loops.

Rule processing is governed by rule priorities and a decision strategy. The `PolicySetComponent` aggregations specify a priority value for each of the contained rules. Priorities are used to define the order in which the rules are evaluated. They are local in the sense that they only define the priority with respect to other rules in the same containing set. Consequently, priorities are much easier to assign than if they had a global meaning. Priority values are unsigned integers and zero denotes the lowest priority. Furthermore, a `PolicySet` instance is always characterized by a decision strategy which can be either FirstMatching or AllMatching. FirstMatching means that the rules of the set are evaluated until a rule matching rule has been found. AllMatching means that all the contained rules are evaluated. In both cases, the rules are evaluated in the order of the priority specified in the `PolicySetComponent` aggregation. In the case of a FirstMatching

strategy, a contained policy rule is considered to match if its condition evaluates to TRUE. A contained policy group is considered to match if at least one of its members matches.

All policy rules are represented as subclasses of `PolicyRule` which essentially represents the "**if** *condition* **then** *action*" rule, i.e. the actions are executed if and only if the condition evaluates to TRUE. When rules are nested, the condition of the parent rule is considered a precondition for its subrules. That is, the parent condition is logically ANDed to all subrule conditions. The parent rule is always evaluated first. So if the parent condition evaluates to TRUE, first the parent actions are carried out and only then the subrules are evaluated. If the parent condition evaluates to FALSE, the subrules are not evaluated.

Rule Conditions Conditions may be either individual conditions or compound conditions. Compound conditions are Boolean combinations of other conditions, expressed in either CNF (Conjunctive Normal Form) or DNF (Disjunctive Normal Form), where the individual subconditions may naturally be negated (see Figure 2.12). Also compound conditions can be combined, which allows for arbitrarily nested and complex condition expressions.

There are several classes related to condition aggregation. `PolicyConditionStructure` is a common superclass that represents aggregations of `PolicyConditions`. Its subclass `PolicyConditionInPolicyRule` associates a condition with a policy rule. Another subclass `PolicyConditionInPolicyCondition` is used to logically combine a set of conditions to a compound condition. Compound conditions are represented by the `CompoundPolicyCondition` class. It includes a property `ConditionListType` that specifies if the subconditions are combined in a CNF or DNF format. `PolicyConditionStructure` furthermore contains two properties that identify how the subcondition is to be treated in the logical combination. First, `GroupNumber` indicates in which logical group it belongs. Second, `ConditionNegated` indicates if the condition is negated or not (illustrated in the figure with the $\pm$ sign).

The abstract class `PolicyCondition` represents the top of the inheritance hierarchy for all rule conditions. Four subclasses of it are defined in PCIM Extensions: `VendorPolicyCondition`, `CompoundPolicyCondition`, `SimplePolicyCondition`, and `PolicyTimePeriodCondition`.

`VendorPolicyCondition` is a generic extension mechanism for ven-

$$\text{(CNF)} \qquad \bigwedge_i \underbrace{\left(\bigvee_j \pm c_{ij}\right)}_{\text{group } i} = (\pm c_{11} \vee \pm c_{12} \vee \ldots \vee \pm c_{1n_1}) \wedge \ldots \wedge (\pm c_{m1} \vee \pm c_{m2} \vee \ldots \vee \pm c_{mn_m})$$

$$\text{(DNF)} \qquad \bigvee_i \underbrace{\left(\bigwedge_j \pm c_{ij}\right)}_{\text{group } i} = (\pm c_{11} \wedge \pm c_{12} \wedge \ldots \wedge \pm c_{1n_1}) \vee \ldots \vee (\pm c_{m1} \wedge \pm c_{m2} \wedge \ldots \wedge \pm c_{mn_m})$$

Figure 2.12: PCIM Rule condition normal forms

dor specific extensions. It contains two properties: `ConstraintEncoding` that specifies the encoding and semantics of the extension as an OID and `Constraint` that is an octet string representation of the condition. `CompoundPolicyCondition` represents the concept of compound policies discussed above. `SimplePolicyCondition` represents the elementary Boolean expression of the type "*variable* MATCH *value*". The matching operator is implied by the model, i.e. not explicitly specified in the condition. Its interpretation depends on the bound variable type and the value instance. Variables are discussed in more depth below in section 2.3.1. A `PolicyTimePeriodCondition` is used to express the validity period of a policy. The validity period is specified in terms of five different masks: an overall time range, month of year, day of month, day of week, and time of day masks. Not all of the masks have to be present in a given condition. The validity period of the policy is considered to be the intersection of all the specified masks. A `PolicyTimePeriodCondition` without any masks specified indicates a policy that is always valid.

Rule Actions Like rule conditions, also rule actions can be combined to form compound policy action sequences and can be contained in other actions. Actions are aggregated by subclasses of `PolicyActionStructure`. Its subclasses `PolicyActionInPolicyRule` aggregates the action in a rule, and `PolicyActionInPolicyAction` is used to combine actions into sequences. The sequences are represented by the `CompoundPolicyAction` class. The execution of the actions of a sequence is characterized by three aspects discussed below: sequence ordering, relevance of the ordering, and

execution strategy.

The actions of a sequence are ordered by the `ActionOrder` property in the `PolicyActionStructure` aggregation. The order is represented as an unsigned integer. Lower values mean earlier execution and the special value of zero indicates that the order for that action is of no importance. It is possible for multiple actions to share the same order value. In this case the relative order of the actions sharing the same value is irrelevant and they can be executed in any order. The relevance of the ordering is indicated by the `SequencedActions` property of the `CompoundPolicyAction` class. The three possibilities are:

- Mandatory: The actions must be carried out in the order specified, or they must not be carried out at all
- Recommended: If possible the order should be respected, if it is not possible the order can be changed
- DontCare: The order is of no importance.

Finally, the execution strategy indicates the desired behavior with respect to the success or failure of subactions when compound actions are aggregated by other compound actions or policy rules. The `ExecutionStrategy` property in the `CompoundPolicyAction` and `PolicyRule` classes indicates how errors encountered in the action execution are treated. The three possible values are:

- Do Until Success: Execute subactions until one of them is successful
- Do All: Execute all subactions independent on if they are successful or not
- Do Until Failure: Execute subactions until one of them is unsuccessful.

PCIM Extensions defines three concrete subclasses of `PolicyAction`: `VendorPolicyAction`, `SimplePolicyAction`, and `CompoundPolicyAction`. As was the case with policy conditions, there is a generic extension class `VendorPolicyAction` for representing policy actions that are not explicitly modeled. Also here vendor extensions contain two properties, `ActionEncoding` for defining the encoding and semantics, and an octet string `ActionData` for representing the data itself. `CompoundPolicyActions` represent action sequences and was discussed above. `SimplePolicyAction` is

the counterpart of `SimplePolicyCondition` and indicates the elementary operation of assigning a value to a variable: "SET *variable* TO *value*".

Policy Variables and Values PCIM Extensions introduces the concept of variables and values. Variables can tested against a value in policy conditions and set in policy actions. Variables are divided in two categories: explicit and implicit. The explicit variables are the ones that have been modeled in a CIM schema, i.e. they simply refer to a property of an existing CIM class. On the other hand, implicit variables are defined and evaluated outside of the model.

All policy variables are represented by the abstract `PolicyVariable` class, of which `PolicyExplicitVariable` and `PolicyImplicitVariable` inherit. The `PolicyExplicitVariable` class contains two string format properties `ModelClass` and `ModelProperty` that indicate the class and property name to which the variable refers. Instead the `PolicyImplicitVariable` class only contains a `ValueTypes` property that lists the allowed policy value types that can the variable can contain. All policy value types inherit from the abstract `PolicyValue` class. PCIM Extensions defines some low-level policy value types such as `PolicyIntegerValue` and `PolicyIPv4AddrValue`.

As noted above, the MATCH operator implied in `SimplePolicyConditions` is context-dependent and its interpretation depends on the actual variable and value instances. For example, different interpretation is needed in the following examples: "`DestinationPort` MATCH 80" (integer comparison) and "`SourceIPAddress` MATCH `MyCompany.Com`" (comparing an IP address to another one in the terms of its DNS address).

Furthermore, both variables and values may be multi-fielded. Each field may be either a single value (singleton), a range of values defined by lower and upper bounds, or a set of values. PCIM Extensions defines a set of matching rules independent on the type of variables and values for the different cases when singletons, ranges, or sets are matched against each other.

Roles Writing individual policies for each managed element is a method that does not scale in complex environments. Instead, policies are targeted at roles. A role is a functional characteristic or capability of a managed element. It is used to determine the applicability of a policy to a particular managed element. Consequently, all relevant elements can be managed

with the same policy.

More specifically, roles are textual names for elements sharing a common characteristic. A managed element may have one or more roles that together form the element's role combination. In PCIM terms, a policy with a given target role combination applies to a managed element if and only if the target role combination is a subset of the managed element's role combination. The target role combination of a policy is specified in the `PolicyRoles` property of the `PolicySet` class. Thus, it is inherited both by `PolicyGroups` and `PolicyRules`. When rules and groups are nested, the semantics is that all roles of the containing rules and groups are automatically inherited by the contained subrules and subgroups.

A group of managed elements sharing a common role are represented with the `PolicyRoleCollection` class. The relevant elements are aggregated to it using the `ElementInPolicyRoleCollection` association class. The name of the role in question is indicated by the `PolicyRole` property of `PolicyRoleCollection`. This name can be matched against the `PolicyRoles` array of roles in `PolicySet` when determining if the policy applies to the elements of the collection.

Conflicting policies There may be several policies that apply to the same managed element but have different actions. This can happen either when composite policies consist of conflicting subpolicies, or when the system has several top-level policies that apply to the same target roles. For the case of subpolicies, the `PolicySetComponent` association class has a priority field that resolves this conflict. In the same manner, top-level policy importance is established by the priority present in the `PolicySetInSystem` association class.

Policy Reuse and Repositories Central to PCIM ideology is the reuse of policies. Useful reusable policy "chunks" can be stored as named elements in the policy repository and then shared by different policies. The policy repository is represented by the `ReusablePolicyContainer` class. Using the `ReusablePolicy` association class, any subclass of `Policy` can be stored in a policy container, including policy groups, rules, (compound) conditions and actions, variables, and values. Even `ReusablePolicyContainers` can be nested using the `ContainedDomain` association class.

Representation of PCIM in XML

The DMTF's Web Based Enterprise Management initiative has produced a specification for the representation of CIM in XML [16]. It defines an XML Document Type Definition (DTD) that fully captures the semantic content of MOF files. So, MOF files can be mapped to XML and vice versa. Actually, the xmlCIM specification contains two conceptually different things: representation of CIM information in XML and representation of CIM Operations in XML. CIM Operations are messages that are transported over HTTP between CIM Clients and CIM Servers. According to [18], work is underway to separate the XML representation from the operations specification.

2.4 Policy language models

The following sections present two general-purpose security models that provide an explicit language for the definition of policies.

2.4.1 Ponder

Overview

Ponder is a generic policy framework that was developed by the Policy Research Group of Imperial College in London. It features a policy specification language [14] and a toolkit. Its goal is to support security and management policy specification in large scale distributed systems.

The Ponder language is declarative and object oriented. Everything is represented by an object interface and policies are written in terms of the interface methods. It includes grouping constructs and policy inheritance for scalability in large systems.

Domains

Domains are a central concept in Ponder. A domain is a filesystem-like hierarchy that is used to group objects such as users, resources, services, and devices into categories for management purposes. Policies can then be addressed to domains instead of individual objects. Thus subdomains can inherit policies from parent domains, and new objects added to the domains are automatically subjected to the correct policies. This allows for easier management of large numbers of objects. Following the filesystem

analogy, in Ponder domains correspond to directories and objects correspond to files. Domains may contain both objects and other subdomains. Objects can also belong to several domains.

Domain Scope Expressions are used to refer to parts of the domain hierarchy. The basic feature is the *domain path* which can be used to refer to the objects in a single subdomain, of all the subdomains up to the specified depth, or of the complete recursive subhierarchy. The usual union, intersection, and difference set operations can be used to combine these results. Furthermore, Object Constraint Language (OCL) select and reject collection operations may be called on sets to subsequently select set members that satisfy a given boolean expression.

Basic policies

The basic policy types in Ponder are authorization, refrain, obligation, and delegation policies. Furthermore, meta policies are supported so that application domain specific constraints can be imposed on the acceptable types of policies in the system. Policies can be specified directly as *instances*, or as reusable parametrized template-like policy *types* from which multiple instances can be created by passing the actual parameters.

Authorization Authorization policies are used to implement access control. In Ponder there are two kinds of authorization policies: positive policies which allow and negative policies which forbid carrying out the desired action. Negative policies could be used for example when policies are specified as general rules with some exceptions, or to temporarily disable access.

Authorization policies consist of subject, target, action, and constraint definitions. An authorization policy states that the *subject* is allowed or forbidden to execute *action* on the *target*. Subject and target definitions are defined as domain scope expressions and the action is a list of methods of the target objects. The optional constraint limits the applicability of the policy to situations where the given expressions evaluates to true. The constraint expressions are specified in a subset of OCL and can be time or state based.

Figure 2.13 presents an example Ponder authorization policy that specifies that the color laser printer can be accessed by executives during business hours. It is assumed that executives and printers be properly assigned to the respective parts in the domain hierarchy.

```
inst auth+ bossPrinterPolicy {
    subject /staff/executives;
    target /printers/colorLaser;
    action print();
    when time.between("0900", "1700");
}
```

Figure 2.13: Example Ponder authorization policy

Refrain Refrain policies are similar to negative authorization policies. The difference is that authorization policies are enforced by targets, whereas refrain policies are enforced by subjects. Therefore, refrain policies express the willingness of the subject not to perform the action on the target even if it would otherwise be allowed to do it. Refrain policies can be used instead of negative authorization policies when the targets cannot be trusted to enforce the negative authorization policy, for example if the target does not wish to be protected from the subject.

One example of a refrain policy could be not giving out detailed product information to the public before the product is officially released. Another example could be refraining from calling employees that are on vacation.

Obligation Obligation policies express responsabilities in the system and are basically event-triggered condition-action rules. An obligation policy defines the activities that a subject must perform on a target when a given event occurs. Events can be either internal to the subject, or generated by external monitoring or event services. More complex events can be built from basic events using event expressions. Events can also be defined separately and then be reused in several policies. Several actions may be specified and may be executed sequentially or in parallel. Uses of obligation policies include security event auditing, dynamic network reconfiguration based on network events, and backup scheduling.

Figure 2.14 is an example obligation policy adapted from [12]. This policy is triggered by 3 consecutive loginfail events with the same userid. The security administrator disables the corresponding user and logs the event.

Delegation Delegation policies are used to specify the circumstances under which subjects can delegate their existing access rights to a grantee. In

```
inst oblig loginFailure {
   on 3*loginfail(userid);
   subject s = /securityAdmin;
   target <userT> t = /users->select (t1 | t1.getId() = userid);
   do t.disable()->s.log(userid);
}
```

Figure 2.14: Example Ponder obligation policy

particular, in Ponder the delegation and revocation processes themselves are operations of the runtime environment and are not initiated by policies. Instead, delegation policies are only used to authorize delegations.

Delegation policies come in two forms: positive and negative. Positive delegation policies authorize the delegation and negative policies forbid it. A delegation policy is always associated with an existing authorization policy which specifies the access rights to be delegated. In Ponder, the grantees can further delegate their granted rights if allowed by the delegation policy. This is called *cascaded delegation*.

Delegations can be constrained in several ways. First, the targets and actions of the original authorization policy to be delegated can be limited by redefining them in the delegation policy. This way, actions and targets can be limited to subsets of the originals. Second, validity of delegation can be constrained based on time restrictions, on attributes of subjects, grantees, targets, and actions, and finally the maximum number of allowed cascading delegations can be defined.

Figure 2.15 presents an example Ponder delegation policy that specifies that bosses can delegate their printer access to their assigned secretaries. The policy is based on the authorization policy of Figure 2.13 and inherits its `target` and `action` fields. It is assumed that employees can be queried for the respective supervisors via the `getBoss()` method.

Composite policies

Ponder incorporates several constructs that let related policies be grouped together for scalability and ease of administration in large systems. The composite constructs are *groups*, *roles*, *relationships*, and *management structures*. As was with basic policies, also with composite policies it is possible to define both direct policy instances and parametrized policy types.

```
inst deleg+ secretaryPrinterDelegPolicy {
    subject boss = /staff/executives;
    grantee secr = /staff/secretaries;
    valid secr.getBoss().userid() = boss.userid();
}
```

Figure 2.15: Example Ponder delegation policy

Groups are used to group together policies that are somehow related from the administrator's point of view. They don't have any special semantics and exist only for policy organization and reuse. Roles are used to gather together policies that relate to a certain organizational position such as a manager or an administrator. All policies in a role share the same subject domain which is defined in the containing role. Relationship composite policies describe relationships between parties. They are very similar to groups and roles, except that in the future Ponder is expected to include syntax for specifying the interaction protocol between the related parties [12]. Management structures are used to describe organizational units such as departments and branch offices that share common roles and relationships.

Additional reusability is attained by allowing composite policy types to specialize other policy types by extending them. Specialized policies automatically inherit all policies from the base policy. Existing policies may be overridden and new ones may be added.

Deployment model

The overall Ponder deployment architecture is depicted in Figure 2.16 which is adapted from [19].

In Ponder, administrators create and edit policies using an *administration tool*. The administration tool includes a policy compiler that transforms the policies to policy classes. These classes are stored in a *policy server*. These classes are instantiated to create *policy control objects* that act as a central point for managing operations of the respective policy.

A *domain service* keeps track of objects in the domain hierarchy and is responsible for evaluating subject and target sets at runtime. Because the domains to which policies refer can change dynamically, the domain server maintains a list of references to all installed policies that apply to

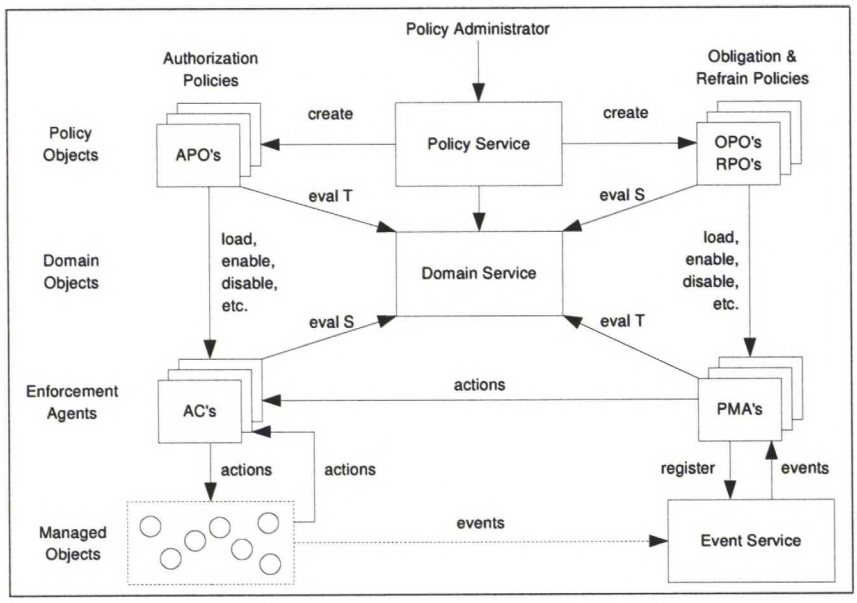

Figure 2.16: Ponder deployment model

each subdomain. When the domain hierarchy changes, the relevant policy objects are notified of the change.

Policy objects do not directly enforce the policies but this responsibility is left to enforcement agents. In case of authorization policies the enforcement agent is the *access controller* of the target object and for obligation and refrain policies it is subject's *policy management agent*. When policies are *loaded*, policy objects distribute *enforcement classes* to the enforcement agent. Authorization enforcement objects are distributed to all policy targets while obligation and refrain policies are distributed to all policy subjects.

Policy management agents of obligations policies register themselves to receive certain events from an *event server*. The *event service* collects system events and notifies the registered event subscribers so that obligation policies can be triggered.

Policy conflicts

Policies may be written by many individuals, and they may cover aspects from different areas of management. Since policy specification in complex systems is not easy, human errors also often cause inconsistencies in the policies. Furthermore, policies can be specified in different languages and

at different levels of abstraction. The refinement of these diverse policies can lead to conflicts in the low level representations. Policy conflicts can spring up in various situations. Policy conflicts have been studied in more detail in [30].

Modality conflicts arise when there are several policies that apply to the same situation and have different outcomes, or modalities. An example of a modality conflict is given by two policies the first of which says that users are not allowed to format hard disks and the second one says the system administrators are allowed to do it. This leads to a modality conflict when it needs to be decided if system administrators are allowed to format hard disks (given that system administrators are also considered users). *Resource conflicts* emerge when the policy controlled targets are not able to support the demands imposed by the policies. For example, policy authors have allocated more network bandwidth than can be provided by the hardware. This conflict may not be trivial to detect if the allocations only apply given sophisticated conditions, or if a change occurs in the underlying hardware (e.g. adding or removing a link).

Some conflicts can be detected at policy specification time, whereas others can only be detected at runtime. For example, policies may have conditions determining their applicability that depend on system state, which is only known at runtime. Some conflicts are generic in nature and can be detected by simply analyzing the policy structure. Others are applications specific and their resolution requires sophisticated knowledge about the application domain. Examples of these are conflict of duty, conflict of interest, and self management conflicts. Some of these conflicts can be detected by encoding application domain knowledge in meta policies. These are policies that constrain the creation of other policies.

Policy conflicts must be resolved somehow, since conflicts may lead the system to a undefined and insecure state. In general, it would be preferable that even runtime conflicts be resolvable without human intervention.

One way to resolve conflicts is to modify the offending policies to work around the conflicts. This is often undesirable, though, because modifying the policies may have adverse effects elsewhere in the system. This is the case especially when there is a generic policy and an exception to it. Devising workarounds may also be difficult and may unnecessarily obscure the original intention of the policy. Another way of avoiding conflicts is to assign explicit priorities to the policies. This method is known to be difficult in a large scale. For authorization and access control policies it could be

specified that negative policies always have precedence. In the end, there is no single method that would be applicable to all situations.

2.4.2 XACML

Overview

XACML (eXtensible Access Control Markup Language) is a general purpose access control policy language specified in [45] by Organization for the Advancement of Structured Information Standards (OASIS). Its goal is to enable different access control systems of large scale systems to interoperate by defining a common standard XML based authorization language. It is designed to be usable in environments where policies are authored by multiple parties such as different administrators and departments.

The XACML authorization model closely resembles the IETF PEP/PDP model [70]. A subject wants to access a resource that is controlled by a PEP (Policy Enforcement Point). PEP outsources the access control decision by sending an *authorization decision request* to its PDP (Policy Decision Point). PDP evaluates the request and returns a *Permit* or *Deny* response.

XACML specifies two languages. The first one is for writing authorization policies. The second one is a language for exchanging authorization requests and responses between PEP and PDP. Both of them are specified using XML Schema. For an introduction to XML Schema see [69].

Authorization model

Below there is a list of the major events that occur when XACML is used for access control (adapted from [45]). The steps are also illustrated in Figure 2.17.

1. *PAPs* write *policies* and *policy sets* and make them available to the *PDP*. These *policies* or *policy sets* represent the complete policy for a specified *target*.

2. The access requester sends a request for access to the *PEP*.

3. The *PEP* sends the request for *access* to the *context handler* in its native request format, optionally including *attributes* of the *subjects*, *resource* and *action*. The *context handler* constructs an XACML request *context* in accordance with steps 4, 5, 6 and 7.

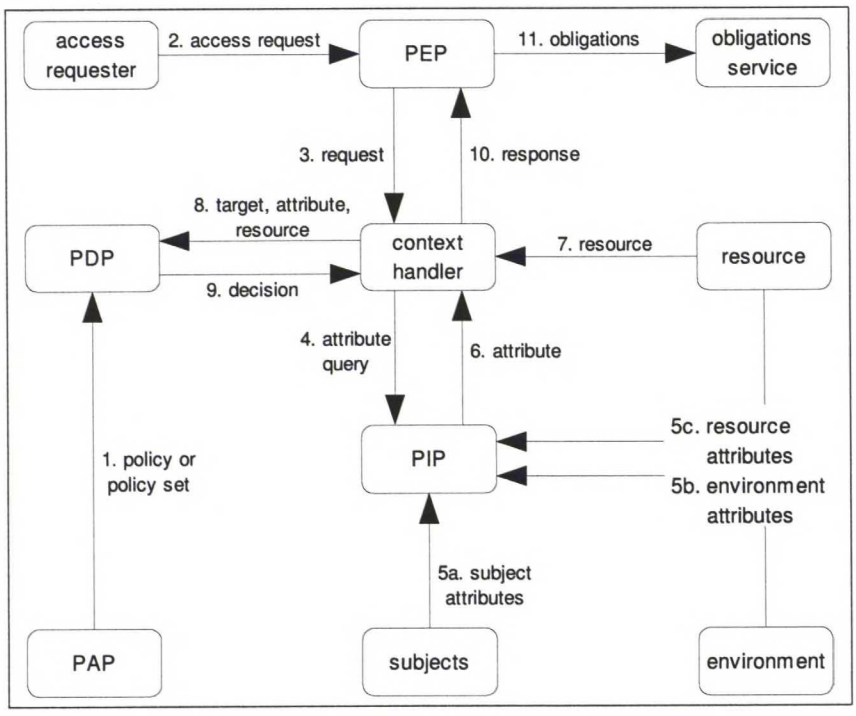

Figure 2.17: XACML Data Flow

4. *Subject*, *resource* and *environment attributes* may be requested from a *PIP* (Policy Information Point).

5. The *PIP* obtains the requested *attributes*.

6. The *PIP* returns the requested *attributes* to the *context handler*.

7. Optionally, the *context handler* includes the *resource* in the *context*.

8. The *context handler* sends a *decision request*, including the *target*, to the *PDP*. The *PDP* identifies the *applicable policy* and retrieves the required *attributes* and (optionally) the *resource* from the *context handler*. The *PDP* evaluates the *policy*.

9. The *PDP* returns the response *context* (including the *authorization decision*) to the *context handler*.

10. The *context handler* translates the response *context* to the native response format of the *PEP*. The *context handler* returns the response to the *PEP*.

11. The *PEP* fulfills the *obligations*.

12. (Not shown) If *access* is permitted, then the *PEP* permits *access* to the *resource*; otherwise, it denies *access*.

Policy elements and evaluation

The focal XACML concept is the *Policy* which forms the basis of an authorization decision. Policies consist of *Rules*, which are the fundamental elements that can be evaluated. Rules cannot be used for authorization decisions in isolation, but are always contained in a Policy. *PolicySets* are grouping constructs that can include other Policies and PolicySets. Included policies can be either specified inline or by reference. The relationships between the language components as specified in [45] are illustrated in Figure 2.18.

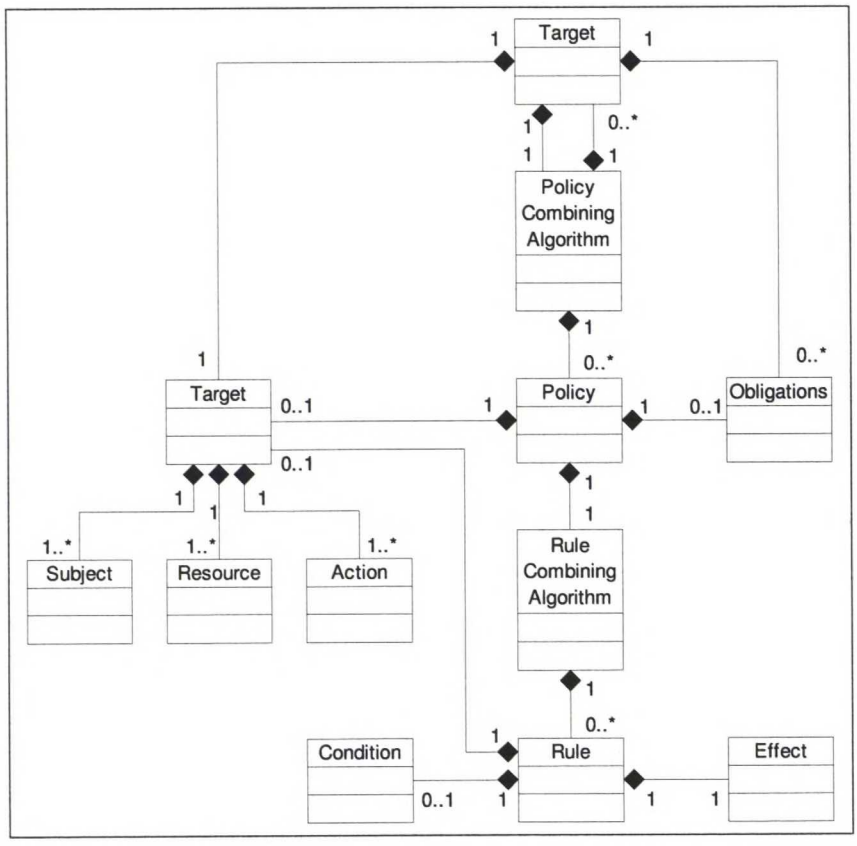

Figure 2.18: XACML Language Model

The PDP must be able to find the policies that are applicable to the decision request given to it. *Targets* can be explicitly specified for Rules,

Policies, and PolicySets. A Target defines the set of resources, subjects, and actions to which these will apply. The PDP can find applicable policies by comparing the information available in the request context to the targets of available policies. If a rule does not contain a target, it inherits the target of its containing Policy. A target evaluates to "Match" if all resources, subjects, and actions match or to "No-match" if they don't. If some of the attributes referenced by the target cannot be obtained, the evaluation result depends on the MustBePresent attribute of the attribute. If it is TRUE, the target evaluates to "Indeterminate", and otherwise to "No-match".

Rules are always associated with an *Effect*, which defines the effect of the rule if it is evaluated to TRUE. It can be either "Permit" or "Deny". Furthermore, rules may optionally contain a *Condition* which is a boolean expression that further limits the rule's applicability. If the condition is absent, it implicitly evaluates to TRUE.

Rule evaluation results in a value that can be "Permit", "Deny", "NotApplicable", or "Indeterminate". If the rule's target matches and its condition is TRUE, the rule's value is what was specified in its effect. A rule results "Indeterminate" if either its target or condition results "Indeterminate". Finally, a result of "NotApplicable" is returned if its target doesn't match or the condition evaluates to false.

Attributes are an essential concept in XACML. They are named values that describe properties of subjects, actions, resources, and the environment of the decision request. XACML attributes are typed and may contain multiple values. Examples of attributes are subject role memberships, subject email addresses and the time of day environment attribute.

Attributes can be referenced by *attribute designators* or *attribute selectors*. Designators specify a name and a type and can refer to subjects, resources, actions, and the environment of the request context. Selectors allow attribute lookup by any XPath query. An example use of selectors is that policies can be specified in terms of the resource contents, if the resource itself is an XML document.

When a PEP sends a decision request to the context handler, it includes any relevant attribute information in the request. In addition, the PDP may request further information which is obtained by the context handler. This information provided in the request context is then compared to the attribute values in its policies in order to make the decision. XACML provides standard operators and functions such as numerical, set, and boolean operators that can be used in formulating the matching expressions.

Rules contained in policies may evaluate to different values. Likewise, policy sets consist of other policies and policy sets which may evaluate to different values. In XACML *Rule Combining Algorithms* are used to determine the decision result of a policy given the evaluation results of its rules, and *Policy Combining Algorithms* are used to determine the value of a policy set given the value of its contained policies. XACML allows users to define their own combining algorithms, and furthermore includes the following standard algorithms:

- *Deny-overrides*: If any included component evaluates to "Deny", the combination results "Deny".
- *Permit-overrides*: If any included component evaluates to "Permit", the combination results "Permit".
- *First applicable*: The result of the combination is the result of the first applicable component.
- *Only-one-applicable*: (Policies only) If there are not any applicable subpolicies, "NotApplicable" is returned or if there are more than one, "Indeterminate" is returned. Otherwise the combination takes the value of the one and only matching policy or policy set.

Policies and policy sets may optionally be associated with *Obligations*. Obligations are always given a "FulfillOn" attribute that specify whether they should be applied when the containing policy evaluates to "Permit" or "Deny". When a PDP evaluates policies or policy sets that contain obligations whose "FulfillOn" attribute matches the policy result, the obligation is returned to the PEP. If the decision result is "Permit", the PEP is responsible for enforcing every obligation returned. If there are obligations that it doesn't understand, it must deny access. If the policy result is "Deny", PEP is only responsible for fulfilling the obligations it understands. Implementations are not required to support obligations since they are an optional feature of XACML.

An example policy

Appendix A contains a complete example XACML policy taken from Sun's XACML Implementation Programmer's Guide [65]. It is an imaginary server login policy. First, the policy defines a Target section that limits the validity of the policy to resources whose AttributeId is SampleServer.

After the Target come two rules. `LoginRule` is the heart of the policy. It has a Target which indicates that the rule applies to server login actions, and a Condition that uses the environment's `current-time` attribute, time comparison functions, and the boolean AND function to specify a validity interval of 9:00 to 17:00. If the target and condition match, the rule permits the action because its Effect attribute is set to Permit. Finally, the policy includes the `FinalRule` which guarantees that access is denied if other rules do not match.

Chapter 3

Analysis of Security Models

This chapter distills the knowledge accumulated in the presentations of the various security models and security policy systems in previous chapters. Section 3.1 summarizes the context in which security models and policies come into being. Section 3.2 proposes a rough classification of different models. Section 3.3 combines the different models to a unifying model and Section 3.4 compares the different models to the unifying model.

3.1 Security modeling context

Security models and policies are rooted in the context of an organization, such as a commercial company, university, or a public sector institution. While policies could in principle be used in small single user systems, their real benefits can only be reaped in large organizations. Standardization of security models and policy languages also paves the way for security policies spanning organizational boundaries.

3.1.1 Organizational security engineering

The need for security policies is ultimately imposed by business level objectives of the organization. This requirements chain is illustrated in Figure 3.1.

All activity in an organization aims to reach *business objectives*. To protect business critical assets from potential threats, organizations' top executives identify and continuously update their security requirements, or *security policy objectives*, that declare the intent of protecting the identified resources from unauthorized use [63].

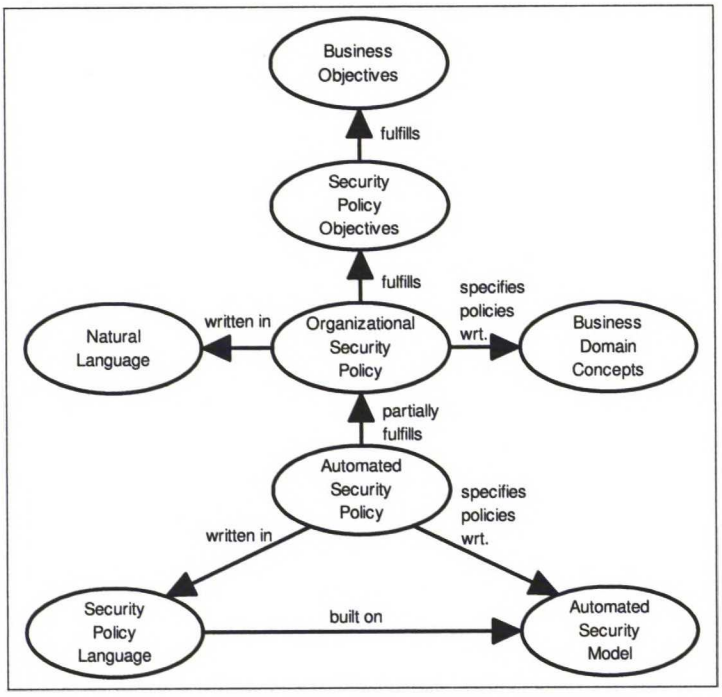

Figure 3.1: Security requirements engineering context

In order to achieve the security policy objectives in practice an *organizational security policy* is devised. It is a guide written in natural language for the individuals in the organization. It consists of rules and practices that must be followed in performing routine activities to meet the specified security policy objectives. Like security policy objectives, also the organizational security policy is fundamentally dependent on the nature of the organization and its business domain. For example, combinations of different industry branches, business models, and organization sizes will imply different security objectives and policies.

Automated security policy is the counterpart of organizational security policy in computer systems. It is a collection of rules that unambiguously instruct computers which system operations are authorized and which ones are not. Its purpose is to enforce the organizational security policy in computer systems. However, as pointed out in [63], computer systems cannot enforce it completely. For example, since a computer cannot tell whether information is classified, it cannot prevent users from entering classified data into unclassified documents.

In order to be understandable by computers, automated security poli-

cies must be written using special purpose languages and tools. Ideally, all security related policies would be managed using a single unified policy authoring system that would automatically take care of refining high-level policies into low-level configuration and distributing it to managed devices. However, as pointed out in the first chapter, in most current real world systems policies are "written" by manually configuring lots of heterogenous and incompatible devices and systems.

3.1.2 Models, policies, and policy languages

In literature, the term *security model* is used with different meanings. In the context of traditional models such as Bell-LaPadula, Biba, Clark-Wilson, and Brewer-Nash a security model consists of both a domain model and a built-in policy. For example, the domain model of Bell-LaPadula and Biba models consist of document objects with sensitivity labels, subjects with clearances, and access operations. Bell-LaPadula has a built-in mandatory multi-level security confidentiality policy whereas Biba has a corresponding integrity counterpart. Similarly, the Brewer-Nash model imposes a built-in policy that a consultant may not deal with more than one client in any competitive sector. While not really policies, also the Clark-Wilson principles of well-formed transactions and separation of duty are also built in the model. Here it is noteworthy that these *traditional models focus on the policy and then build the required modeling infrastructure around it*. Instead, Role Based Access Control is a pure modeling concept. It is completely policy neutral and does not even provide any facilities for authoring them.

On the other hand, all recent security policy language systems abandon the idea of built-in policies. They can be seen as a evolutionary step beyond the traditional models since they allow policies to be adapted to organization's needs. This is in fact a very natural development step since computer systems are increasingly used in diverse environments, unlike thirty years ago when they mainly served military and governmental institutions.

To clarify discussion and to distinguish between the aforementioned concepts the following terms are suggested for the purposes of the rest of this work:

- *Security model*: A conceptual model of all security related aspects of a computer system. May or may not support policy based management.
- *Security system*: An implementation of a security model ready for use by an organization.

- *Security policy system*: A security system that supports policy based management.
- *Security policy*: A configuration of a security policy system expressed in a security policy language and written by security administrators that satisfies the security requirements of their respective organization, or a fixed security policy implied by the security model itself.
- *Security policy language*: A language which is exclusively constructed to allow writing security policies with respect to a specific security model.

These definitions explicitly highlight the conceptual distinction between a security policy and a security model, and the distinction between a security policy language and security model.

While conceptually different, it is obvious that a security policy language is necessarily tightly related to the security model whose policies it expresses. In fact, in all security policy language systems presented in this work the languages exactly match their underlying models. However, RBAC is a clear counterexample since it provides modeling constructs but does not have a corresponding policy language.

These definitions also leave vague a lot about what a security model actually contains. This issue is dealt with in more detail in the following section.

3.2 Model classification

Each of the security models presented above has been designed with a distinct objective in mind. Therefore the models describe slightly different target domains and are of different nature. A rough classification of the models is illustrated in Figure 3.2 and is presented in more detail below.

At a high level, the models have been classified into two categories: proper models and metamodels. While the separation between the two is not absolutely strict, the distinctive factor is the security policy. For our purposes, *a model is classified as a proper model if and only if it either has a clearly defined policy or provides a mechanism that allows one to be defined.* Models without this property are classified as *metamodels*.

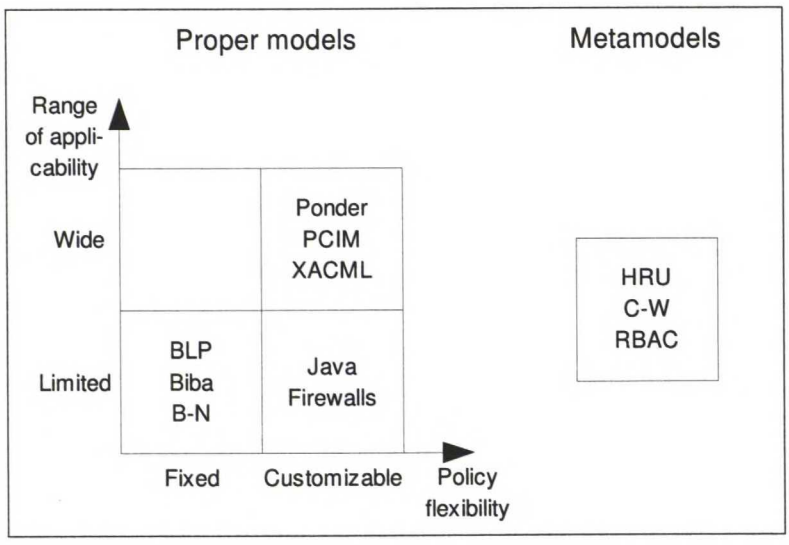

Figure 3.2: Security model classification

3.2.1 Metamodels

Metamodels describe desirable properties of proper models. Because of the lack a security policy, they cannot be applied as such in real-life situations. Rather they provide guidelines and best practices that can be applied in the design of proper models and their respective security policies. While the Role Based Access Control and Clark-Wilson models are intended for real use, the Harrison-Ruzzo-Ullman model only serves the purpose of providing a foundation for proving the undecidability of the Safety Problem.

3.2.2 Proper models

The proper models have been further classified with respect to two dimensions: range of applicability and policy flexibility. In the traditional Bell-LaPadula, Biba, and Brewer-Nash models security administrators have little power to modify the security policy. The only available means of customization in these models is the assignment of security labels, but fundamentally the security policy always remains the same.

On the other hand, in the other proper models the security policy can be customized in a much more fine-grained manner. Ponder and XACML provide explicit languages for expressing policies. PCIM does not provide a language but rather a framework for modeling them. The Java security architecture has a flexible policy model that is not tied to any specific policy

language, but the reference implementation comes with one such language. There do not exist any significant standards for expressing firewall policies.

The second dimension, range of applicability, measures the fitness for purpose of the models in different domains of application. The fixed-policy models are closely tied to the traditional problem of protecting sensitive documents in military and commercial organizations. The models for Java and firewalls have been designed to address only their respective needs. On the other hand, Ponder, Policy Common Information Model, and XACML have been explicitly designed to be applicable in a wide range of environments. Hence they can be used in a wealth of diverse applications and devices, such as enterprise applications, web browsers, networked printers, electronic door locks, and mobile phones. It is also worth observing that they are generic enough to implement the security policies of the other security models.

To summarize, we have chosen policy definition, policy flexibility, and range of applicability as the criteria for model classification. These criteria classify models into four coherent groups:

- Traditional models (Bell-LaPadula, Biba, Brewer-Nash)
- Pragmatic models (Java security, Firewalls)
- Flexible models (Ponder, Policy Common Information Model, XACML)
- Metamodels (Harrison-Ruzzo-Ullman, Clark-Wilson, Role Based Access Control)

We feel that these criteria provide useful insight into the nature of different security models (see Section 3.5 for more discussion about this). However, obviously there exist several other criteria that can serve as an alternative basis for classification. Some other potential model properties that could serve as classification criteria are listed below.

- Main security goal: confidentiality, integrity
- Intended industry: military, government, commercial, public sector, domestic
- Formality and mathematical verifiability
- Support for large-scale centralized management

- Support for a formal policy language
- Age

3.3 Unifying model

This section constructs a single conceptual unifying model out of the different properties and capabilities of different security models that were presented in the preceding chapters. The purpose of the unifying model is to serve as a yardstick against which the different models are then analyzed in Section 3.4.

3.3.1 Security model capabilities

The core of a security policy system is the security model. Its design influences just about every other component in the system. Based on the different models presented in the previous chapters, four conceptually separate categories of related capabilities can be distinguished:

- Domain capabilities
- Policy capabilities
- Administration capabilities
- Policy deployment capabilities

These categories are discussed in the following sections.

Domain capabilities

Domain capabilities are features that facilitate description of security relevant properties of business domain concepts. Usually domain concepts are mapped to OO-like objects that contain attributes and methods. In principle, domain objects can be modeled at any level of abstraction. For example, a security model to be enforced inside an operating system would define the model in terms of e.g. files and sockets, whereas a business application would model entities such as business transactions, accounts, processes, documents, or employees. Some models limit the modeled objects to a certain domain, whereas others are flexible enough to facilitate modeling a large variety of different problem domains. Furthermore, to facilitate management of complex systems a domain object model usually provides mechanisms to organize similar entities into groups or hierarchies.

Policy capabilities

Policy capabilities determine which kinds of policies can be defined. The most fundamental distinction is between fixed and custom policies. Many traditional models impose a fixed security policy that cannot be changed. More recent models allow organizations to devise their own policies.

Fundamentally, there are two types of policies. Authorization policies are the essence of security systems. Furthermore, some systems also support obligation policies for automating management tasks.

An authorization policy essentially consists of rules that have two parts: *applicability range* and an *authorization decision*. An authorization decision simply states if access is permitted or denied. The applicability range specifies the conditions under which the decision is effective. That is, if the conditions match the current access context, the access will be permitted or denied according to the policy decision. Otherwise the decision is ignored and other potential available policy rules are examined. The applicability range has been traditionally defined with respect to subject or object identities and the operation to be performed on the object. This approach is not always the most suitable one and is likely the main reason for the wide acceptance of the RBAC model. More generally, conditions could refer to anything that can be programmatically accessed and manipulated. Examples are other properties than identity of the parties involved in the operation such as company affiliation or a service account balance, state of the system such system load or Defcon level, or environmental factors such as time or date.

An obligation policy specifies actions that must be carried out when a given *event* occurs in the system. They are mainly used for automating tasks that come up in managing large systems and are not necessarily security related. Examples include logging certain events, performing backups, reacting to failing components, and on-demand software installation. Unlike authorization policies which are invoked upon an access request, obligation policies are triggered by external events. Execution of the actions can be further restricted by using conditions like the ones in authorization policy applicability ranges.

Since both authorization and obligation policies are usually written in a declarative rather than procedural manner, it is possible for many policies to be applicable in a given situation. This is even likely if a large number of policies exist or if policies have been written by different people. For authorization policies, different policies can potentially have conflicting au-

thorization decisions and hence the system must be able to decide which of the policies takes priority. For obligation policies, it may be desirable that when several policies match the event condition either none, just one, or all obligations be carried out. In general, both authorization and obligations policies need a *conflict resolution strategy* that determine system behavior when several (potentially conflicting) policies are applicable. Some possible strategies are assigning absolute numerical priorities to policies, specifying "A takes priority over B" relations, or preferring the policy whose condition somehow matches the current context most closely. Furthermore, authorization policy models need a *default policy decision* that stipulates the authorization decision in the case that there are not any matching policies.

Administrative capabilities

To ease policy administration most policy models provide programming language-like mechanisms for writing policies. In a spirit similar to computer programs that are composed of subroutines and modules, policies are usually built out of rules and can be further aggregated into policy sets. Other well-known and analogous reuse mechanisms include parametrized templates and inheritance.

For some of the security models presented, it is possible to formally verify if an automated security policy satisfies the requirements set by the corresponding organizational security policy. The more formal and simple the model is, the easier it is to verify its properties. Thus the recent models with flexible policies do not have this capability.

In early security models little attention was paid to administration. Either it was completely ignored or it was supposed that there was a central security officer that somehow took care of administrating the system. As one indication of the static approach of the early models is the Bell-LaPadula tranquility principle which even requires that once fixed the security labels of subjects and objects never change.

Given the dynamic nature of many organizations it is not reasonable to expect that the static administration model work in practice. Consequently, recent developments such as Administrative RBAC have put a lot of effort on studying how administration should be carried out. Firstly, it is taken granted that policies are written by several cooperating administrators. Secondly, it is also seen imperative that in large organizations administrative authority and responsibility can be decentralized because a central security team cannot possibly keep track of all the needs of different

departments, projects, and teams. Instead it should be possible to delegate authority to the local groups of people where also the most knowledge of their security needs resides. One basic but useful way of decentralization is to delegate administrative rights along the lines of organizational hierarchy from higher positions to lower ones.

While delegation can be used to decentralize administration tasks, it is also useful outside the administration domain. Namely, temporary delegation can provide support for backing up absent people and facilitating collaboration [71]. Naturally, administrators should be able to specify policies that restrict the conditions under which rights can be delegated.

Finally, any policies administrators write should themselves be potentially controllable by other policies. These policies are called metapolicies since instead of describing organization's security policies they describe what kind of policies can be written. Metapolicies are a useful concept but it seems that their applications have not yet been studied extensively.

Policy deployment capabilities

The most notable policy deployment capability is the way policies are disseminated. In the provisioning ("push") model, policies are automatically distributed to all managed elements in advance and they are evaluated locally in the managed element. In the outsourcing ("pull") model policy enforcement points do not compute policy decisions themselves but instead outsource the decision to a policy decision point ("pull" model) at the time of access.

In addition to the separation of PEP and PDP a complete security policy system may be viewed as containing other conceptual services as well. A Policy Administration Point consists of the hardware and software used for authoring policies. A policy repository stores system policies and is not necessarily integrated with the PDP. An event service may be needed to mediate obligation policy triggers. A context service may be needed to provide both generic environment and organization specific state attributes for policy evaluation. Depending on the security model and its implementation any of these components may be integrated together or be distributed.

3.3.2 Unifying the capabilities

We define the *unifying model* as a hypothetical security model comprising all the capabilities below (which were discussed above):

- Domain capabilities
  - Modeling arbitrary objects
  - Grouping constructs
- Policy capabilities
  - Custom policies
  - Authorization policies
  - Obligation policies
  - Arbitrary properties
  - Unambiguous policies
- Administrative capabilities
  - Policy reuse
  - Policy verifiability
  - Delegation of authority
  - Metapolicies
- Policy deployment capabilities
  - Policy outsourcing
  - Policy provisioning

There are obvious dependencies between some of the capabilities. For example, a security model that imposes a fixed policy model obviously does not provide mechanisms for policy reuse. Furthermore, some of the capabilities are contradictory. For example, a model that supports custom policies with respect to arbitrary objects is unlikely to be formally verifiable. Hence the unifying model remains purely fictitious and exists only for the purposes of the following section.

3.4 Comparison with the unifying model

From Section 3.2 it can be seen that there is remarkable dissimilarity between some of the models. Consequently, as it is an artificial unification of the different models, the unifying model includes some capability areas that do not make a lot of sense when compared to some of the models.

3.4.1 Metamodels

Harrison-Ruzzo-Ullman

Domain capabilities The HRU model is oriented towards the traditional scenario where the access by programs acting on behalf of subject users to object documents is controlled. It is not well suited for modeling other access control scenarios. Being just a mathematical tool it does not provide tools for grouping subjects or objects but rather each of them is managed individually by directly modifying the elements in the access matrix.

Policy capabilities This model does not have the notion of a policy or policy customization. Nonetheless, an implementation of the HRU command set can be seen as implementing a policy. The HRU model deals with authorization policies but not obligation policies. Access control is exclusively based on subject and object identities and the access matrix. Hence arbitrary properties cannot be used in policy decisions. Since the access matrix gives unambiguous decisions, there is no need for conflict resolution mechanisms.

Administrative capabilities Because of the lack of the notion of a policy this model does not provide support for policy reuse, delegation of authority, or metapolicies. On the other hand, it has a strong focus on policy verifiability (here a policy is considered to be equivalent to the implementation of the command set). The authors specifically prove that in its general form the policy verification problem is undecidable, but some restricted types of policy are decidable.

Policy deployment capabilities Given the lack of explicit policies, there are no mechanisms for policy provisioning or outsourcing.

Clark-Wilson

Domain capabilities The Clark-Wilson model does not explicitly state the nature of the data items whose protection it addresses. Nonetheless, it is quite obvious that the model has been designed to protect business documents from malicious employees. Hence it is not suitable for radically different environments such as network packet traversal or Java privileged operation invocation. The model does not provide any explicit mechanisms for grouping data items.

Policy capabilities In the Clark-Wilson model, the relation of the E2 rule forces a representation of access control decisions but not how the policies leading to them are managed. Clark-Wilson deals with authorization policies but not obligation policies. The E2 relation is based on identities of users and data items, and thus arbitrary properties cannot be referred to in authorization decisions. Policy decisions are always unambiguous since all access is denied by default and only the accesses in the E2 relation are allowed.

Administrative capabilities Since it lacks the notion of a policy this model does not provide support for policy reuse, delegation of authority, or metapolicies. If a system and its administrative personnel fulfill all the Clark-Wilson rules, the system will always remain secure. However, the rules are generic high-level requirements that leave a lot of room for interpretation in real implementations. Therefore, verifying the safety of a system based on this model reduces to verifying the conformance of the implementation to the model rules. Obviously, there is no general algorithm for this.

Policy deployment capabilities Because of the lack of explicit policies, there are no mechanisms for policy provisioning or outsourcing.

Role Based Access Control

Domain capabilities By concentrating on roles rather than user identities, RBAC gains flexibility with respect to more traditional models. Nonetheless, it is still oriented towards user access control and not applicable to radically different situations such as network packet traversal or Java privileged operation invocation. RBAC provides roles as a convenient grouping mechanisms of subjects, but does not provide any mechanisms for grouping managed objects.

Policy capabilities Since RBAC does not have a policy concept, the custom policies and conflict resolution capabilities do not apply. The model addresses authorization policies, but not obligation policies. Since permissions are directly related to subject roles, arbitrary properties cannot be referred to in policy decisions.

Administrative capabilities Because of the lack of the notion of a policy, the capabilities of policy reuse, policy verifiability, and metapolicies do not apply. However, the Administrative RBAC model provides explicit constructs for delegating responsibility of subsets of the role hierarchy to subordinates.

Policy deployment capabilities Given the lack of explicit policies, the mechanisms for policy provisioning or outsourcing do not apply.

3.4.2 Proper models

Bell-LaPadula, Biba, and Brewer-Nash

Domain capabilities As these models are closely tied to the problem of protecting sensitive documents, they are not well suited to modeling arbitrary objects. For example, it would be unwieldy to model network packet traversal or Java privileged operation invocation with lattice-based security label assignment. Multilevel security sensitivity levels and categories provide a basic but inflexible mechanism for grouping domain objects. The same applies to the mechanism of tagging documents with their respective corporation and industry in the Brewer-Nash model.

Policy capabilities All of these models incorporate a fixed authorization policy. None of these models are able to express obligation policies or any custom policies whatsoever. The fixed policies are defined with respect to built-in security labels and arbitrary object properties can not be referred to. Also, the predefined policy is always unambiguous and hence there is no need for a conflict resolution mechanism.

Administrative capabilities A fixed policy eliminates the need for a policy reuse mechanism. In the same vein, metapolicies are not needed for controlling policy authoring. Since these models are very simple, their safety properties can be formally proved. As a side note, the resulting proofs may be of little practical value, since the built-in fixed policy does not usually match the organization's security policy. These models do not provide any mechanisms for delegating authority other than direct manipulation of the security labels. Consequently, delegation is a difficult concept to implement with these models.

Policy deployment capabilities Since the policy is fixed in these models, there is no need for policy provisioning or outsourcing mechanisms.

Java security

Domain capabilities Being tightly related to the authorization of privileged Java operations, the policy model is not suitable for modeling arbitrary objects. Codebase URLs and principal properties provide basic mechanisms for grouping subjects and the various permission classes can provide different levels of grouping for objects.

Policy capabilities The `AccessController` Java class provides an interface for querying authorization policy decisions. Users may provide their own implementations, which allows for arbitrary policies to be constructed. In addition, the reference implementation includes a sample security policy language. There are no interfaces for obligation policies. Since arbitrary programs can be used to compute authorization decisions, any arbitrary subject and object properties can be referred to in the policies. Also the responsibility of conflict resolution is offloaded to the policy implementation logic. The reference model has a "deny everything except when explicitly allowed" logic which cannot cause conflicts between policy rules.

Administrative capabilities Since an implementation of the whole authorization framework can be provided by users, the administrative capabilities completely depend on the implementation. As for the reference implementation, the policy language does not support any policy reuse mechanisms, and because of the complex and dynamic nature of the authorization framework and the policies, mathematical verification of safety properties is most likely very difficult if not impossible. There is no explicit support for delegation of authority or metapolicies.

Policy deployment capabilities As with administrative capabilities, also the deployment capabilities are dependent on the implementation. The Java software itself always outsources the decision to the access controller. However, depending on the implementation the policies may be distributed from a policy authoring point to access controllers in advance, or local access controllers may still delegate the decision to a remote decision point.

Firewalls

There are neither widely accepted policy language standards nor authoring mechanisms for firewall security policies. The comparison below assumes the traditional method of manually writing low-level vendor-dependent packet filter rules directly on each firewall device.

Domain capabilities Like in the Java security model, also firewall policies are closely related to the traffic control problem and hence not usable for modeling other types of access control. Since network packets are of transient nature, there is no need for grouping specific objects in advance. However, conditions that match packet data fields can be used to define the applicability of firewall rules to groups of packets at a time.

Policy capabilities Firewalls must necessarily allow security administrators a lot of flexibility in specifying their policies since every organizations' network is unique. Firewall policies are concerned about the authorization of traversing network packets and there is no need for obligation policies. However, logging rules can also be seen as one form of obligation policies. While most policy rules are based on IP addresses and TCP/UDP ports, in principle policies could also be based on a wide variety of criteria such as IPSec credentials, time of day, a defcon level, or arbitrary data encapsulated in the packet's payload. Practically taken all firewall policies sidestep the conflict resolution issue by requiring that policy rules be ordered and giving priority to the rule that comes first.

Administrative capabilities Most low-level firewall policy languages do not have explicit mechanisms to facilitate policy reuse, delegation of authority, or metapolicies. Since the packet filter concept is relatively simple and uniform among different firewall models, it is possible to verify some properties of firewall policies as discussed in Section 2.2.2.

Policy deployment capabilities Traditionally, policies are either authored directly in the firewall or remotely and then distributed to the firewall. For performance reasons, the outsourcing model is not appropriate for firewalls.

Ponder

Domain capabilities Ponder is a generic framework that is capable of modeling diverse environments and arbitrary objects. It provides a domain mechanism that facilitates grouping both subjects and objects.

Policy capabilities Custom policies can be written in the Ponder policy language which supports both authorization and obligation policies. All entities are modeled as OO-like objects, whose properties can freely be referred to in policy decisions. In Ponder, it is possible to specify conflicting policies. The policy language does not any have mechanisms for assigning priorities to rules to avoid unambiguities. However, there are tools for detecting conflicts that help administrators to resolve the conflicts by manually adjusting the policy.

Administrative capabilities The Ponder policy language provides policy inheritance and composite policies as mechanisms for policy reuse. It also has policy types for delegation policies and metapolicies. Since it is a generic-purpose language and policies may form conflicts, formal verification of a policy is very difficult or even impossible.

Policy deployment capabilities The Ponder deployment model is based on the concept of automatically distributing policies and their enforcing agents to managed objects. While in principle it would be possible to build external policy decision points that would compute policy decisions for enforcement points, the Ponder model does not address this possibility.

XACML

Domain capabilities Like Ponder, XACML is a generic language that can express policies with respect to any objects. While XACML does not provide explicit grouping constructs, its ability to refer to arbitrary properties of objects can be used as a versatile mechanism of selecting a group of objects. For example, to implement the RBAC role concept, objects could be labeled with a set of strings that identify its roles. Or, to implement the Ponder domain concept, there could be a central domain entity that would track all entities, or all entities could be directly labeled with a set of domain paths to which they belong.

Policy capabilities XACML allows administrators complete freedom in customizing policies. The basic function of the language is to describe authorization policies. There is also limited support for carrying out obligations as a result of evaluating authorization policies. As noted above, arbitrary properties of objects can be used as a criterion in policy decisions. Rule and policy combining algorithms are used to unambiguously define the final outcome of a policy that consists of subpolicies and subrules. Hence inside a specific policy, conflicts are not possible. However, the question of conflicts between separate top level policies remains apparently unanswered in XACML.

Administrative capabilities The fact that in XACML policies may contain other policies and rules as parts facilitates policy reuse. There are neither mechanisms for delegating policy authoring responsibility nor for specifying other types of metapolicies. As was the case with Ponder, the expressiveness of XACML makes formal verification very hard.

Policy deployment capabilities XACML deployment is focused on the outsourcing concept and the provisioning model is not addressed. Nonetheless, in principle there is nothing in the language itself that would prevent a system being built that would automatically distribute policies to enforcement points in advance.

Policy Common Information Model

Domain capabilities PCIM has been designed to be able to model any objects (the CIM Schema already provides definitions of a wide range of managed objects). Policies are targeted to roles that define groups of managed objects. Furthermore, the possibility of referring to arbitrary object properties allows for more fine-grained selection of objects.

Policy capabilities Even the basic `SimplePolicyCondition` and `SimplePolicyAction` structures allow a wide variety of policies to be described. Furthermore, PCIM allows the specification of vendor specific conditions and actions. PCIM focuses on obligation policies. There are no explicit mechanisms for modeling access control conditions and decisions in particular, but in principle these could be implemented easily as new condition and action extensions. As discussed before, arbitrary object properties can

be used in policies. Conflicts both in subpolicies inside composite policies and top-level policies are resolved by explicit priority properties.

Administrative capabilities Policy reuse is possible through composite policies and the storage of named policy chunks in the repository of reusable policies. As was the case with Ponder and XACML, the expressiveness of PCIM implies that verification is very hard. There are no mechanisms for delegation or metapolicies.

Policy deployment capabilities The PCIM framework has been designed to fit the IETF outsourcing based management scenario. Again, there is nothing in PCIM itself that would prevent policies to be provisioned to enforcement points in advance.

3.4.3 Summary

A summary of the capabilities of the various models with respect to the unifying model is illustrated at a rough level in Figures 3.3 and 3.4. The legend used is: N = No, Y = Yes, P = Partial, NA = Not Applicable.

	HRU	C-W	RBAC	BLP	Biba	B-N
DOMAIN CAPABILITIES						
Modeling arbitrary objects	N	N	P	N	N	N
Grouping constructs	N	N	P	P	P	P
POLICY CAPABILITIES						
Custom policies	NA	NA	NA	N	N	N
Authorization policies	Y	Y	Y	Y	Y	Y
Obligation policies	N	N	N	N	N	N
Arbitrary properties	N	N	N	N	N	N
Unambiguous policies	NA	NA	NA	Y	Y	Y
ADMINISTRATIVE CAPABILITIES						
Policy reuse	NA	NA	NA	NA	NA	NA
Policy verifiability	P	P	NA	Y	Y	Y
Delegation of authority	NA	NA	Y	N	N	N
Metapolicies	NA	NA	NA	N	N	N
POLICY DEPLOYMENT CAPABILITIES						
Outsourcing	NA	NA	NA	NA	NA	NA
Provisioning	NA	NA	NA	NA	NA	NA

Figure 3.3: Summary of model capabilities (part 1)

	Java	FW	Ponder	XACML	PCIM
DOMAIN CAPABILITIES					
Modeling arbitrary objects	N	N	Y	Y	Y
Grouping constructs	P	P	Y	P	P
POLICY CAPABILITIES					
Custom policies	Y	Y	Y	Y	Y
Authorization policies	Y	Y	Y	Y	P
Obligation policies	N	N	Y	P	Y
Arbitrary properties	Y	P	Y	Y	Y
Unambiguous policies	NA	P	N	P	Y
ADMINISTRATIVE CAPABILITIES					
Policy reuse	N	N	Y	Y	Y
Policy verifiability	N	P	N	N	N
Delegation of authority	N	N	Y	N	N
Metapolicies	N	N	Y	N	N
POLICY DEPLOYMENT CAPABILITIES					
Outsourcing	Y	N	N	Y	Y
Provisioning	N	Y	Y	N	N

Figure 3.4: Summary of model capabilities (part 2)

3.5 Common core

As stated in Section 1.2, the objective of this thesis was to study and compare currently available security models and find out what is common to all of them, i.e. their common core. As can be seen from the summary in Section 3.4.3, none of the analyzed capabilities are common to all the models. (The authorization policy capability is an obvious exception because this work is about security models and that capability was chosen only to differentiate between authorization and obligation policies.) Therefore, based on the findings of this chapter, the models when considered all together form a heterogeneous group whose common core is practically non-existent. Nonetheless, when the models are divided to classes as was illustrated in Figure 3.2, they form four largely homogeneous groups.

The traditional models (Bell-LaPadula, Biba, and Brewer-Nash) are very similar to each other and in Figure 3.3 they share exactly the same capabilities. This is natural since the Biba model is directly based on BLP, and all the models can be seen as special cases of Lattice Based Access Control.

The class of pragmatic models (Java security, Firewalls) is also quite co-

herent. Both models are designed for a specific purpose but allow policies to be customized. Both models allow policies to be specified in a policy language. However, there are no standards for firewall policy languages, and the Java policy language is only an example implementation. It is evident that neither of the models has been designed for large scale complex environment since there is little support for policy administration.

The flexible models (Ponder, PCIM, XACML) have designed to overcome the main weakness of the earlier models: restricted domain of applicability. All these three models are generic frameworks that can be utilized in highly diverse environments. Consequently, all the models also provide a flexible policy model or language. These models are also the only ones to support obligation policies. Finally, Ponder has more advanced administrative features than the two other models.

Metamodels (Harrison-Ruzzo-Ullman, Clark-Wilson, Role Based Access Control) are rather similar from the capability viewpoint. Since they lack the concept of policy, they do not require many of the policy and administrative capabilities that the rest of the models provide. Harrison-Ruzzo-Ullman serves only as a theoretical devices as a means for the Safety Problem undecidability proof, while Clark-Wilson and Role Based Access Control provide guidelines for building real security models.

Chapter 4

Discussion and conclusions

4.1 Model applicability

One important part of application security is transport security, i.e. ensuring secure communication between applications. The advent of the Internet has given rise to many technologies for securing communication over an untrusted network. Some examples include Transport Layer Security, IPSec, and Secure Shell. Another aspect is internal application security logic, i.e. the authorization of privileged operations inside business applications. Currently there are no standard mechanisms for internal application security. In other words, there exist widely adopted standards for *inter*-application security, but not for *intra*-application security.

As stated in Section 1.2, the secondary objective of this thesis was to assess the applicability of the different security models to serve as general-purpose security models and policy languages in a policy-based security architecture. This conceptual architecture is basically an application of the generic IETF policy framework architecture for security management. The IETF architecture is illustrated in Figure 4.1 (as cited by [66]). There is also a related Framework for Policy-based Admission Control specification [70]. While it is primarily designed for Quality of Service management, it is suitable for access control as well.

As discussed earlier in Section 1.4.2, policy based management can reduce and automate administrative work considerably. But there are also advantages for system and software providers. The separation of application logic and authorization logic improves maintainability not only for system administrators, but for software developers as well. Furthermore, reuse of policy language and policy decision engine components can cut

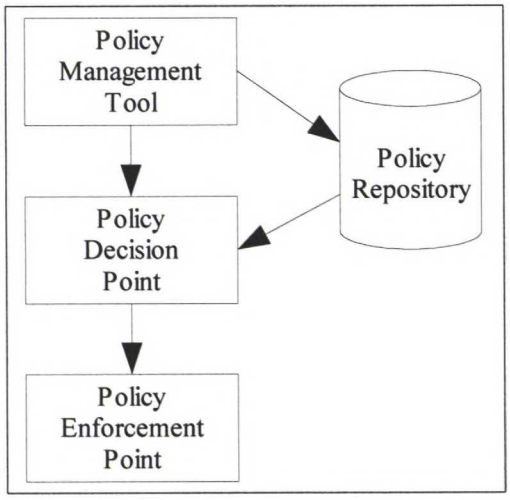

Figure 4.1: IETF Policy Framework Architecture

down an development effort and costs. Some APIs are emerging for the application and security logic separation, such as Java Authentication and Authorization Service and Generic Authorization and Access-control API (GAA-API) [52].

In order to be reusable in a wide variety of application scenarios, a security model and the corresponding policy language should above all be generic and flexible. This requisite precludes all the models covered in this thesis except PCIM, Ponder, and XACML. All of these three models are designed to be flexible and are therefore suitable as a part of the policy-based security architecture. Of these three, Ponder and XACML provide a policy language while PCIM only provides an object-oriented conceptual policy model and does not currently provide a language for policy representation. However, a policy language should be developed for PCIM for it to be widely adoptable.

A lot of work still lies ahead to realize the policy-based security architecture in practice, but the most important problems seem to have been conceptually solved. Most of the remaining work consists integration and tool support.

4.2 On security model research hurdles

Security model and security policy research would most likely benefit greatly from better understanding real organizations from the security

point of view. At least two areas should be studied more extensively: security policies and organizational issues.

Having abundant research material on real security requirements and security policies would provide good indication of what capabilities future security models should provide. Designing security models without real security requirements is analogous to designing software without a requirements specification. Unfortunately, most organizations either do not have explicit written security policies or they are confidential. Therefore real security policies are hard to come by, which makes it difficult to evaluate the usefulness of security models in fulfilling real security requirements.

Also studying the common patterns of organizational structure and behavior would allow models to be designed that better support the organization. As is the case with any other tool, also security models should fit the organization and not vice versa. This sort of multi-disciplinary research would require people skilled in technology, organizational sciences, and psychology.

4.3 Summary

The primary objective of this thesis was to study different security models and find out their common core. Based on the analysis, when viewed as a whole the models are largely disparate and have few common features. Nonetheless, there are significant similarities between some of the models, which lead to a division of models in traditional, pragmatic, flexible, and metamodels.

The secondary objective was to assess the applicability of the different security models to facilitate the development of secure business applications in the future. The Ponder and XACML models were found to be flexible enough to provide a building block for a policy-based security architecture. PCIM would also be suitable if there was a policy representation language.

Appendix A

XACML Example Policy

```
<Policy PolicyId="SamplePolicy"
        RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:
                            rule-combining-algorithm:first-applicable">

  <!-- This Policy only applies to requests on the SampleServer -->
  <Target>
    <Subjects>
      <AnySubject/>
    </Subjects>
    <Resources>
      <Resource>
        <ResourceMatch
         MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
          <AttributeValue
           DataType="http://www.w3.org/2001/XMLSchema#string">
            SampleServer
          </AttributeValue>
          <ResourceAttributeDesignator
           DataType="http://www.w3.org/2001/XMLSchema#string"
           AttributeId=
             "urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>
        </ResourceMatch>
      </Resource>
    </Resources>
    <Actions>
      <AnyAction/>
    </Actions>
  </Target>

  <!-- Rule to see if we should allow the Subject to login -->
  <Rule RuleId="LoginRule" Effect="Permit">

    <!-- Only use this Rule if the action is login -->
```

```
<Target>
  <Subjects>
    <AnySubject/>
  </Subjects>
  <Resources>
    <AnyResource/>
  </Resources>
  <Actions>
    <Action>
      <ActionMatch
       MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
        <AttributeValue
         DataType="http://www.w3.org/2001/XMLSchema#string">
          login
        </AttributeValue>
        <ActionAttributeDesignator
         DataType="http://www.w3.org/2001/XMLSchema#string"
         AttributeId="ServerAction"/>
      </ActionMatch>
    </Action>
  </Actions>
</Target>

<!-- Only allow logins from 9am to 5pm -->
<Condition FunctionId="urn:oasis:names:tc:xacml:1.0:function:and">
  <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:
                     time-greater-than-or-equal">
    <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:
                       time-one-and-only">
      <EnvironmentAttributeDesignator
       DataType="http://www.w3.org/2001/XMLSchema#time"
       AttributeId="urn:oasis:names:tc:xacml:1.0:environment:
                    current-time"/>
    </Apply>
    <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#time">
      09:00:00
    </AttributeValue>
  </Apply>
  <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:
                     time-less-than-or-equal">
    <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:
                       time-one-and-only">
      <EnvironmentAttributeDesignator
       DataType="http://www.w3.org/2001/XMLSchema#time"
       AttributeId="urn:oasis:names:tc:xacml:1.0:environment:
                    current-time"/>
    </Apply>
```

```
        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#time">
          17:00:00
        </AttributeValue>
      </Apply>
    </Condition>

  </Rule>

  <!-- We could include other Rules for different actions here -->

  <!-- A final, "fall-through" Rule that always Denies -->
  <Rule RuleId="FinalRule" Effect="Deny"/>

</Policy>
```

Appendix B

CIM Public key certificate MOF

```
// ==================================================================
// PublicKeyCertificate
// ==================================================================
[Version("2.6.0"), Description(
  "A Public Key Certificate is a credential "
  "that is cryptographically signed by a trusted Certificate "
  "Authority (CA) and issued to an authenticated entity "
  "(e.g., human user, service,etc.) called the Subject in "
  "the certificate and represented by the UsersAccess class. "
  "The public key in the certificate is cryptographically "
  "related to a private key that is to be held and kept "
  "private by the authenticated Subject.  The certificate "
  "and its related private key can then be used for "
  "establishing trust relationships and securing "
  "communications with the Subject.  Refer to the ITU/CCITT "
  "X.509 standard as an example of such certificates.") ]
class CIM_PublicKeyCertificate : CIM_Credential {

  [Propagated("CIM_CertificateAuthority.SystemCreationClassName"),
    Key, MaxLen(256), Description("The scoping System's CCN.") ]
  string SystemCreationClassName;

  [Propagated("CIM_CertificateAuthority.SystemName"),
    Key, MaxLen(256),Description("The scoping System's Name.") ]
  string SystemName;

  [Propagated("CIM_CertificateAuthority.CreationClassName"),
    Key, MaxLen(256), Description("The scoping Service's CCN.") ]
  string ServiceCreationClassName;

  [Propagated("CIM_CertificateAuthority.Name"),
    Key, MaxLen(256), Description("The scoping Service's Name.") ]
  string ServiceName;
```

```
  [Key, MaxLen(256), Description(
    "Certificate subject identifier.") ]
  string Subject;

  [MaxLen(256), Description(
    "Alternate subject identifier for the Certificate.") ]
  string AltSubject;

  [Octetstring, Description("The DER-encoded raw public key.") ]
  uint8 PublicKey[];
};
```

Bibliography

[1] Marshall D. Abrams and David Bailey. Abstraction and Refinement of Layered Security Policy. In Marshall D. Abrams, Sushil Jajodia, and Harold J. Podell, editors, *Information Security: An Integrated Collection of Essays*. IEEE Computer Society Press, Los Alamitos, California, USA, 1995.

[2] Ross Anderson. *Security Engineering - A Guide to Building Dependable Distributed Systems*, chapter 7.3. John Wiley & Sons, 2001.

[3] R. W. Baldwin. Naming and grouping privileges to simplify security management in large databases. In *Proceedings of 1990 IEEE Symposium on Security and Privacy*, pages 116–132, Oakland, CA, USA, May 1990.

[4] A. Bandara, E. Lupu, and A. Russo. Using Event Calculus to Formalize Policy Specifications and Analysis. In *IEEE 4th International Workshop on Policies for Distributed Systems and Networks*, pages 26–39, June 2003.

[5] Y. Bartal, A. Mayer, K. Nissim, and A. Wool. Firmato: a novel firewall management toolkit. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 17–31, 1999.

[6] David Bell. Concerning "Modeling" of Computer Security. In *Proceedings of 1988 IEEE Symposium on Security and Privacy*, pages 8–13, Oakland, CA, USA, April 1988.

[7] David Bell and Leonard LaPadula. Secure Computer Systems: Unified Exposition and Multics Interpretation. Technical Report ESD-TR-75-306, MITRE, 1975.

[8] Steven M. Bellovin. Distributed Firewalls. *;login: magazine*, pages 39–47, November 1999.

[9] Ken Biba. Integrity Considerations for Secure Computer Systems. Technical Report MTR-3153, Mitre Corporation, Bedford MA, USA, 1977.

[10] David F. C. Brewer and Michael J. Nash. The Chinese Wall security policy. In *Proceedings of 1989 IEEE Symposium on Security and Privacy*, pages 206–214, Oakland, CA, USA, May 1989.

[11] David D. Clark and David R. Wilson. A Comparison of Commercial and Military Computer Security Policies. In *Proceedings of 1987 IEEE Symposium on Security and Privacy*, pages 184–194, Oakland, CA, USA, April 1987.

[12] N. Damianou. *A Policy Framework for Management of Distributed Systems*. PhD thesis, Imperial College of Science, Technology and Medicine, London, UK, February 2002.

[13] N. Damianou, A. Bandara, M. Sloman, and E. Lupu. A Survey of Policy Specification Approaches. Technical report, Department of Computing Imperial College of Science Technology and Medicine London, 2002.

[14] N. Damianou, N. Dulay, E. Lupu, and Morris Sloman. Ponder: A Language for Specifying Security and Management Policies for Distributed Systems, October 2000.

[15] R. Darimont and A. Lamsweerde. Formal refinement patterns for goal-driven requirements elaboration. In *Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, pages 179–190, 1996.

[16] Distributed Management Task Force. Specification for the Representation of CIM in XML, May 2002. DSP0201.

[17] Distributed Management Task Force Policy Working Group. CIM Policy Model White Paper, March 2002. DSP0108.

[18] Distributed Management Task Force Technical Committee. The Common Information Model, January 2003. Technical Note.

[19] N. Dulay, E. Lupu, M. Sloman, and N. Damianou. A Policy Deployment Model for the Ponder Language. In *Proc. IEEE/IFIP International Symposium on Integrated Network Management*. IEEE Press, May 2001. extended version.

[20] Pasi Eronen and Jukka Zitting. An expert system for analyzing firewall rules. In *Proceedings of the sixth Nordic Workshop on Secure IT-Systems (NordSec 2001)*, pages 100–107, November 2001.

[21] David Ferraiolo and Richard Kuhn. Role-Based Access Control. In *Proceedings of the NIST-NSA National (USA) Computer Security Conference*, pages 554–563, 1992.

[22] Dieter Gollmann. *Computer Security*, chapter 4. John Wiley & Sons, 1999.

[23] Li Gong. JavaTM 2 Platform Security Architecture, version 1.2. Technical report, Sun Microsystems, 2002. Available online: <`http://java.sun.com/j2se/1.4.2/docs/guide/security/spec/security-spec.doc.html`>[Referenced 2004-09-01].

[24] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in Operating Systems. *Communications of the ACM*, 19(8):461–471, August 1976.

[25] A. Herzberg, Y. Mass, J. Mihaeli, D. Naor, and Y. Ravid. Access control meets public key infrastructure, or: assigning roles to strangers. In *Proceedings of 2000 IEEE Symposium on Security and Privacy*, pages 2–14, 2000.

[26] Thomas Hildmann and Jörg Barholdt. Managing trust between collaborating companies using outsourced role based access control. In *Proceedings of the fourth ACM workshop on Role-based access control*, pages 105–111, 1999.

[27] Sotiris Ioannidis, Angelos D. Keromytis, Steve M. Bellovin, and Jonathan M. Smith. Implementing a distributed firewall. In *Proceedings of the 7th ACM conference on Computer and communications security*, pages 190–199, 2000.

[28] Charlie Lai, Li Gong, Larry Koved, Anthony Nadalin, and Roland Schemers. User Authentication and Authorization in the Java Platform. In *15th Annual Computer Security Applications Conference*, pages 285–290. IEEE Computer Society Press, 1999.

[29] Butler W. Lampson. Protection. In *Proc. 5th Princeton Conf. on Information Sciences and Systems*, March 1971.

[30] E. Lupu and M. Sloman. Conflicts in Policy-Based Distributed Systems Management. *IEEE Transactions on Software Engineering*, 25(6):852–869, November 1999.

[31] M. Masullo and S. Calo. Policy management: an architecture and approach. In *IEEE First International Workshop on Systems Management*, pages 13–26, April 1993.

[32] Alain Mayer, Avishai Wool, and Elisha Ziskind. Fang: A Firewall Analysis Engine. In *Proceedings of 2000 IEEE Symposium on Security and Privacy*, pages 177–187, May 2000.

[33] Terry Mayfield, J. Eric Roskos, Stephen R. Welke, and John M. Boone. Integrity in Automated Information Systems. Technical Report C-TR-79-91, National Computer Security Center, September 1991.

[34] John McLean. A Comment on the "Basic Security Theorem" of Bell and LaPadula. *Information Processing Letters*, 20:67–70, February 1985.

[35] John McLean. Reasoning About Security Models. In *Proceedings of 1987 IEEE Symposium on Security and Privacy*, pages 123–131, Oakland, CA, USA, April 1987.

[36] John McLean. The Specification and Modeling of Computer Security. *IEEE Computer*, 23(1):9–16, 1990.

[37] John McLean. *Encyclopedia of Software Engineering*, chapter Security Models. Wiley Press, 1994.

[38] Lynn M. Meredith. A Summary of the Autonomic Distributed Firewalls (ADF) Project. In *Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX 03)*, volume 2, pages 260–265, April 2003.

[39] J. B. Michael, V. L. Ong, N. C. Rowe, and Naval Postgraduate School. Natural-Language Processing Support for Developing Policy-Governed Software Systems. In *39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems*, 2001.

[40] Jonathan Millen. Editor's Preface to the Bell-LaPadula Model. *Journal of Computer Security*, 4(2/3), 1996.

[41] J. Moffett and M. Sloman. Policy Hierarchies for Distributed Systems Management. *IEEE Journal on Selected Areas in Communications*, 11:1404–1414, 1993.

[42] M. Casassa Mont, A. Baldwin, and C. Goh. POWER Prototype: Towards Integrated Policy-Based Management. Technical report, HP Laboratories Bristol, October 1999.

[43] B. Moore. Policy Core Information Model (PCIM) Extensions. RFC 3460, Internet Engineering Task Force, January 2003.

[44] Matunda Nyanchama and Sylvia Osborn. Modeling mandatory access control in role-based security systems. In *Proceedings of Database security IX : status and prospects*, pages 129–144, 1996.

[45] OASIS XACML Technical Committee. eXtensible Access Control Markup Language (XACML) Version 1.0. Standard, OASIS, February 2003.

[46] Sejong Oh and Seog Park. An Improved Administration Method on Role-Based Access Control in the Enterprise Environment. *Journal of Information Science and Engineering*, 17(6):921–944, November 2001.

[47] Sejong Oh and Ravi Sandhu. A Model for Role Administration Using Organization Structure. In *Proceedings of the seventh ACM symposium on Access control models and technologies*, pages 155–162, 2002.

[48] Sylvia Osborn, Ravi Sandhu, and Qamar Munawer. Configuring role-based access control to enforce mandatory and discretionary access control policies. *ACM Transactions on Information and System Security*, 3(2):85–106, 2000.

[49] Charles Payne and Tom Markham. Architecture and Applications for a Distributed Embedded Firewall. In *Proceedings of the 17th Annual Computer Security Applications Conference (ACSAC 2001)*, pages 329–336, December 2001.

[50] Najam Perwaiz. Structured Management of Role-Permission Relationships. In *Proceedings of the sixth ACM symposium on Access control models and technologies*, pages 163–169, 2001.

[51] Fausto Rabitti, Elisa Bertino, Won Kim, and Darrell Woelk. A model of authorization for next-generation database systems. *ACM Transactions on Database Systems*, 16(1):88–131, 1991.

[52] Tatyana Ryutov and Clifford Neuman. Access Control Framework for Distributed Applications. Internet Draft draft-ietf-cat-acc-cntrl-frmw-05, USC/Information Sciences Institute, November 2000.

[53] Vipin Samar. Unified login with pluggable authentication modules (PAM). In *Proceedings of the 3rd ACM conference on Computer and communications security*, pages 1–10, 1996.

[54] Ravi Sandhu. Role Activation Hierarchies. In *Proceedings of the third ACM workshop on Role-based access control*, pages 33–40, 1998.

[55] Ravi Sandhu, Venkata Bhamidipati, and Qamar Munawer. The ARBAC97 Model for Role-Based Administration of Roles. *ACM Transactions on Information and System Security*, 2(1):105–135, 1999.

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

[57] Ravi Sandhu, David Ferraiolo, and Richard Kuhn. The NIST Model for Role-Based Access Control: Towards a Unified Standard. In *Proceedings of the fifth ACM workshop on Role-based access control*, pages 47–63, 2000.

[58] Ravi S. Sandhu. A Lattice Interpretation of the Chinese Wall Policy. In *Proceedings of the 15th NIST-NCSC National Computer Security Conference*, pages 221–235, Baltimore, MD, USA, October 1992.

[59] Ravi S. Sandhu. Future Directions in Role-Based Access Control Models. In *Information Assurance in Computer Networks: Methods, Models, and Architectures for Network Security, International Workshop MMM-ACNS 2001, St. Petersburg, Russia, May 21-23, 2001, Proceedings*, volume 2052 of *Lecture Notes in Computer Science*, pages 22–26. Springer, 2001.

[60] Christoph Schuba and Eugene H. Spafford. A Reference Model for Firewall Technology. In *Proceedings of the 13th Annual Computer Security Applications Conference*, pages 133–145, 1997.

[61] M. Sloman, N. Dulay, and B. Nuseibeh. The secpol project: Specification and analysis of security policy for distributed systems. Imperial College, Department of Computing. Available online: <http://www-dse.doc.ic.ac.uk/projects/secpol/SecPol-overview.html>[Referenced 2003-07-09].

[62] Gary W. Smith. *The Modeling and Representation of Security Semantics for Database Applications*. PhD thesis, George Mason University, Fairfax, VA, USA, 1990.

[63] Daniel F. Sterne. On the Buzzword "Security Policy". In *Proceedings of 1991 IEEE Symposium on Security and Privacy*, pages 219–230, Oakland, CA, USA, May 1991.

[64] Sun Microsystems. Java Authentication and Authorization Service (JAAS) Reference Guide for the Java 2 SDK, Standard Edition, v 1.4, August 2001. Available online: <http://java.sun.com/j2se/1.4.2/docs/guide/security/jaas/JAASRefGuide.html>[Referenced 2004-09-01].

[65] Sun Microsystems. *Sun's XACML Implementation Programmer's Guide*, February 2003. Available online: <http://sunxacml.sourceforge.net/guide.html>[Referenced 2004-08-29].

[66] D. Verma. Simplifying Network Administration Using Policy-Based Management. *IEEE Network*, 16(2):20–26, March 2002.

[67] Rene Wies. Using a Classification of Management Policies for Policy Specification and Policy Transformation. In Adarshpal S. Sethi, Yves Raynaud, and Fabienne Fure-Vincent, editors, *Integrated Network Management IV*, volume 4, pages 44–56, Santa Barbara, CA, 1995. Chapman & Hall.

[68] Avishai Wool. Architecting the Lumeta Firewall Analyzer. In *Proceedings of the 10th Usenix Security Symposium*, August 2001.

[69] World Wide Web Consortium. XML Schema Part 0: Primer. W3C Recommendation.

[70] R. Yavatkar, D. Pendarakis, and R. Guerin. A Framework for Policy-based Admission Control. RFC 2753, Internet Engineering Task Force, January 2000.

[71] Xinwen Zhang, Sejong Oh, and Ravi Sandhu. PBDM: A Flexible Delegation Model in RBAC. In *Proceedings of the eighth ACM symposium on Access control models and technologies*, pages 149–157, Como, Italy, 2003.