Patterns for Access Control in Distributed Systems

Nelly Delessy
Florida Atlantic University
Dept of CEECS
Boca Raton, Florida, USA
+1 561 297-3855
ndelessy@fau.edu

Eduardo B. Fernandez
Florida Atlantic University
Dept of CEECS
Boca Raton, Florida, USA
+1 561 297-3466
ed@cse.fau.edu

M. M. Larrondo-Petrie
Florida Atlantic University
College of Engineering
Boca Raton, Florida, USA
+1 561 297-3899
petrie@fau.edu

Jie Wu
Florida Atlantic University
Dept of CEECS
Boca Raton, Florida, USA
+1 561 297-3855
jie@cse.fau.edu

ABSTRACT
Distributed systems introduce a new variety of security threats. The organizations that own those systems must protect their information assets from attacks. To do this we need to start with high-level models that represent the security policies of the institution. We present patterns that derive from traditional models: first, the Policy-Based Access Control which models how to decide if a subject is authorized to access an object according to policies defined in a central policy repository. Then we present implementation-oriented patterns that implement the Access Matrix or RBAC model: The ACL pattern allows control access to objects by indicating which subjects can access an object and in what way. There is usually an ACL associated with each object. The Capability pattern allows control access to objects by providing a credential or ticket to be given to a subject for accessing an object in a specific way. Capabilities are given to the principal.

Categories and Subject Descriptors
D.2.11 [**Software Architectures**]: Patterns

General Terms
Design, Security

Keywords
Security Patterns, access control, software architecture

1. INTRODUCTION
Distributed systems are typically heterogeneous systems that are opened to a wide variety of partners, customers or mobile employees that introduce a new variety of security threats. They are widely used by organizations that must protect their information assets from attacks. Those information assets typically are accessed through services that come in a variety of technologies. It is important to develop systems where security has been considered at all stages of design, which not only satisfy their functional specifications but also satisfy security requirements. To do this we need to start with high-level models that represent the security policies of the institution [1].

Preliminary versions of these papers were workshopped at Pattern Languages of Programming (PLoP) '07 September 5-8, 2007, Monticello, IL, USA. ISBN: 978-1-60558-411-9.

To protect its assets, an organization needs to define security policies, which are high-level guidelines that specify the states in which the system is considered to be secure [2]. These policies need to be enforced by security mechanisms. In large organizations, the policies may be issued by different actors making their management difficult. Moreover, they need to be enforced for a variety of resources.

Furthermore, the nature of distributed systems implies that a subject does not need to be known in advance by the system in order to request access to a resource. The use of credentials including attributes may be sufficient to trust a subject. Policies should be able to capture this aspect. Figure 1 illustrates some patterns used in access control in the context of distributed systems. A pattern diagram shows relationships between patterns (represented by rectangles with rounded corners). The relationships appear as labeled arrows. In this diagram, traditional models, such as the Access Matrix and RBAC (Role-Based Access Control), are represented along with Attribute-Based Access control [3] and Policy-Based Access control. The two latter models are more suitable in the case of distributed systems. All of the models use a Reference Monitor to enforce access decisions. ACL (Access Control List) and Capability are implementation-oriented patterns; they implement the Access Matrix or RBAC model. More specifically for web services, XACML (eXtensible Access Control Markup Language) Access Control Evaluation implements the Attribute-Based Access control pattern and the Policy-Based Access control pattern, and the XACML Policy Language implements the Policy-Based Access control pattern. SAML Authorization Assertion is a kind of Capability.

In this paper we present the following patterns:

- Policy-Based Access Control: models how to decide if a subject is authorized to access an object according to policies defined in a policy repository
- Access Control List: controls access to objects by indicating which subjects can access an object and in what way. There is usually an ACL associated with each object.
- Capability: controls access to objects by providing a credential or ticket to be given to a subject for accessing an object in a specific way.

These patterns are of value to security designers and software developers implementing distributed systems.

Section 2, 3 and 4 respectively present the aforementioned patterns and section 5 concludes the paper.

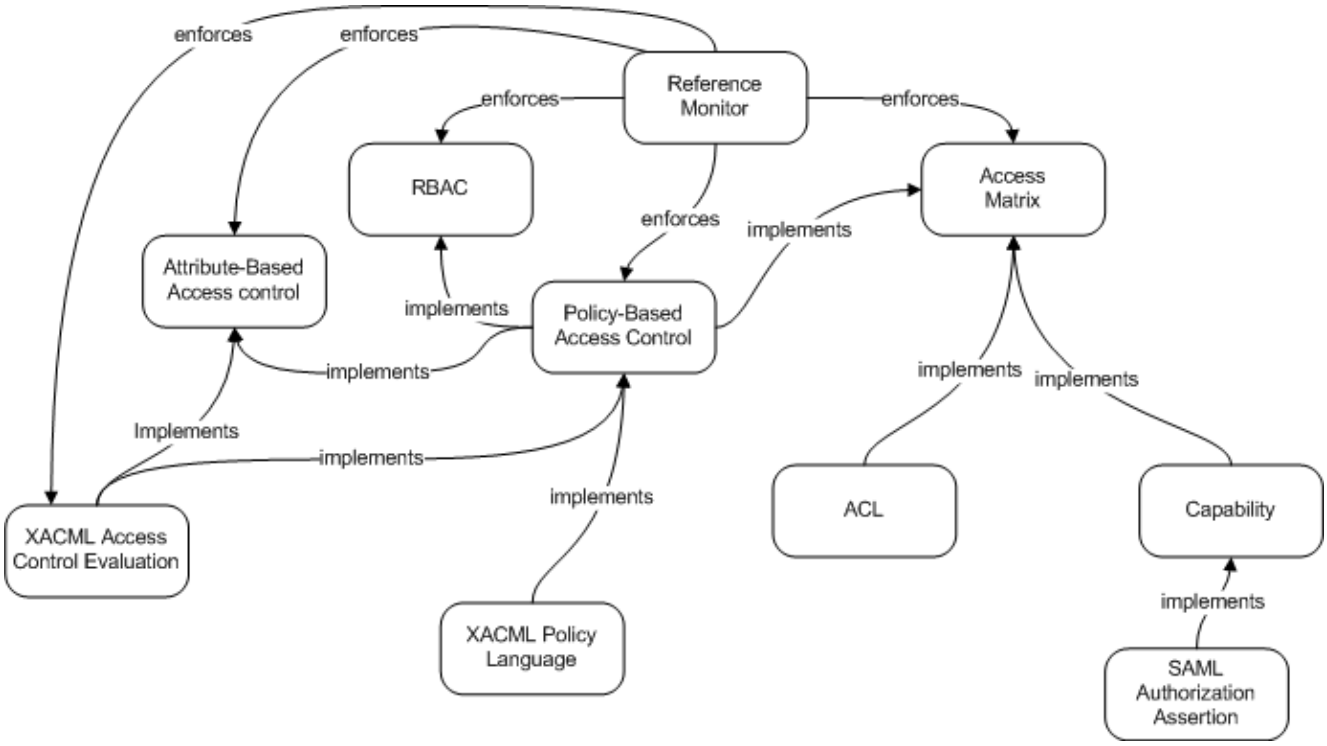

Figure 1. Pattern diagram for access control in web services

2. POLICY-BASED ACCESS CONTROL

The Policy-Based Access Control pattern decides if a subject is authorized to access an object according to policies defined in a central policy repository.

Example

Consider a financial company that provides services to its customers. Their computer systems can be accessed by customers who send orders to the company for buying or selling commodities (stocks, bonds, real estate, art, etc.) via email or through their website. Brokers employed by the company can carry out the orders of the customers by sending requests to the systems of various financial markets, or consult information from financial news websites. Also, a government auditor visits periodically to check for application of laws and regulations.

All of these activities are regulated by policies with various granularities within the company. For example, the billing department can have the rule «only registered customers whose account status is in good standing may send orders», the technical department can decide that «emails with attachments bigger than x Mb won't be delivered», the company security policy can state that «only employees with a "broker" role can access the financial market's web services» and that «only the broker custodian of a customer can access its transaction information», whereas the legal department issues the rule that «auditors can access all transaction information», etc.

All of these policies are enforced by different components of the computer system of the company (email server, file system, web service access control component, and financial application). This approach has several problems: the policies are described in possibly different syntaxes and it is difficult to have a global view of what policies apply to a specific case. Moreover, two policies can be conflicting and there is no way to combine them in a clear way. In summary, this approach could be error-prone and complex to manage.

Context

We consider centralized or distributed systems with a large number of resources (objects). A large number of subjects may access those objects. Rules are defined to control access to objects. The rules defined by the organization are typically designed by different actors (technical, organizational, legal, etc), and each set of rules designed by a specific policy designer can concern overlapping sets of objects and/or subjects. We assume that access requests come from authenticated subjects.

Problem

Enforcing these rules for a particular access request may be complex, and thus error prone, because there is no clear view of what rules to apply to a request.

How can we enforce access control according to the pre-defined rules in a consistent way? The solution to this problem is affected by the following forces:

- Objects may be frequently added or removed

- The solution should be able to implement a wide variety of access control models, such as the Access Matrix, RBAC.
- Malicious users can try to have access to unauthorized objects.
- There should be no direct access to objects, i.e. every request must be mediated.

Solution

Most access control systems are based on the authorization pattern [4], where the access of a subject to an object depends only on the existence of a positive applicable rule. If no such rule exists, then the access is denied. In our case, the situation is more complicated: the existence of a positive applicable rule should not necessarily imply that the access should be granted. All of the rules must be taken into account, and a final decision must be made from the set of applicable rules and some meta-information about the way they should be combined. Part of that meta-information is located in a policy object. This policy object aggregates a set of rules, and specifies how those rules must be combined. For more flexibility about the combination of rules, a composite object regroups the rules into policies and Policy Sets. Basically, policy sets aggregate policies, and includes information about how to combine rules from different policies. In order to easily select all applicable rules, they should be stored in a unique repository for the organization and administered in a centralized way. At access time, all requests are intercepted by Policy Enforcement Points (PEPs), a specific type of Reference Monitor [2]. The repository is accessed by a unique Policy Decision Point (PDP), which is responsible for computing the access decision by cooperating with a Policy Information Point (PIP), which may provide information about the subject or the resource accessed. The rules and policies are administered through a unique Policy Administration Point (PAP). Finally, because rules and policies are designed by different teams, possibly about the same objects and subjects, this scheme does not guarantee that a conflict between rules in different policy components would never occur. In that case, the PDP may have a Dynamic Policy Conflict Resolver to resolve the conflict, which would need to use meta-rules. A complementary Static Policy Conflict Resolver may be a part of the PAP, and should detect conflicts between rules at the time they are entered into the repository.

Structure

Figure 2 illustrates the solution.

A **Subjects'** access requests to particular **Objects** of the system are intercepted by **PEPs,** which are a part of the security infrastructure that is responsible for enforcing the organization **Policy** about this access**. PEPs** query another part of the security infrastructure, the **PDP**, which is responsible for computing an access decision. In order to compute the decision, the **PDP** uses information from a **PIP**, and retrieves the applicable **Policy** from the unique **PolicyRepository**. A **PolicyRepository** stores all of the rules for the organization. It is also responsible for retrieving the applicable **Rules** by selecting those **Rules** whose **subjectDescriptor**, **resourceDescriptor**, and **environmentDescriptor** match the information about the subject, the resource and the environment pulled from the **PIP**, and whose **accessType** matches the required **accessType** from the request. The **PAP** is a unique point for administering the rules. In case the evaluation of the Policy leads to a conflict between the decisions of the applicable Rules, a part of the **PDP**, the **DynamicPolicyConflictResolver,** is responsible for producing a uniquely determined access decision. Similarly, a **StaticPolicyConflictResolver** is a part of the **PAP** and is responsible for identifying conflicting rules within the **PolicyRepository**.

Dynamics

Figure 3 shows a sequence diagram describing the most commonly used case of Request Object Access. The Subject's request for accessing an **Object** is intercepted by a **PEP**, which forwards the request to the **PDP**. The **PDP** can retrieve information about the **Subject**, the **Object** and the current **Environment** from the **PIP**. This information is used to retrieve the applicable **Rules** from the **PolicyRepository**. The **PDP** can then compute the access decision by combining the decisions from the **Rules** forming the applicable **Policy** and it can finally send this decision back to the **PEP**. If the access has been granted by the **PDP**, the **PEP** forwards the request to the **Object**.

Example Resolved

The use of the Policy-Based Access Control pattern a;;ows the company to centralize its rules. Now, the billing department, as well as the technical department, the legal department and the corporate can insert their rules in the same repository, using the same format. The different components of the computer system that used to enforce policies directly (that is, email server, file system, web service access control component, and financial application) just need to intercept the requests and redirect them to the central Policy Decision Point. In order to do that, each of them runs a Policy Enforcement Point, which interfaces with the main Policy Decision Point. The rules could be grouped in the following way: a unique company policy set might include all other policies and express that all policies coming from the corporate should dominate all other policies. Each department would have their own policy, composed of rules from this department, and combined according to each department's policy. Finally, a simple dynamic conflict resolver could be configured to enforce a closed policy in case of conflict. The rules can be easily managed, since they are written to the same repository, the conflicts can be resolved, and there is a clearer view of the company's security policy.

Known Uses

- XACML (eXtensible Access Control Markup Language), defined by OASIS, includes languages for expressing authorization rules and for access decision following this pattern.
- Symlabs Federated Identity Access Manager Federation is an identity management from Symlabs implementing identity federation. Its components include a PDP and PEPs.
- "Components Framework for Policy-Based Admission Control", a part of the Internet 2 project, is a framework for the authentication of network components. It is based on five major components: Access Requestor (AR), Policy Enforcement Point (PEP), Policy Decision Point (PDP), Policy Repository (PR), and the Network Detection Point (NDP).
- XML and Application firewalls [5] also use policies.

SAML (Security Assertion Markup Language) is an XML standard defined by OASIS for exchanging authentication and authorization data between security domains. It can be used to transmit the authorization decision.

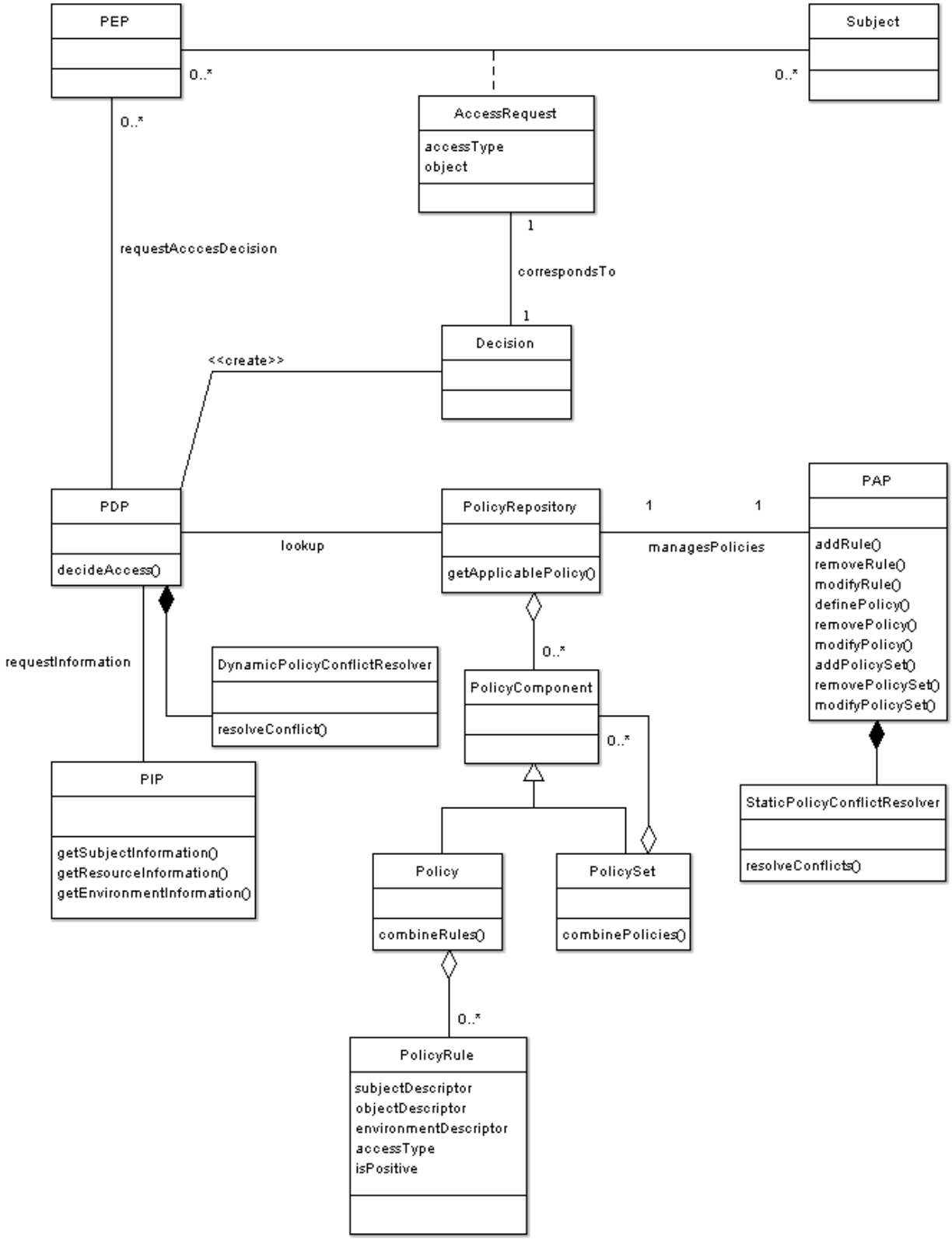

Figure 2. Class diagram for Policy-Based Access Control

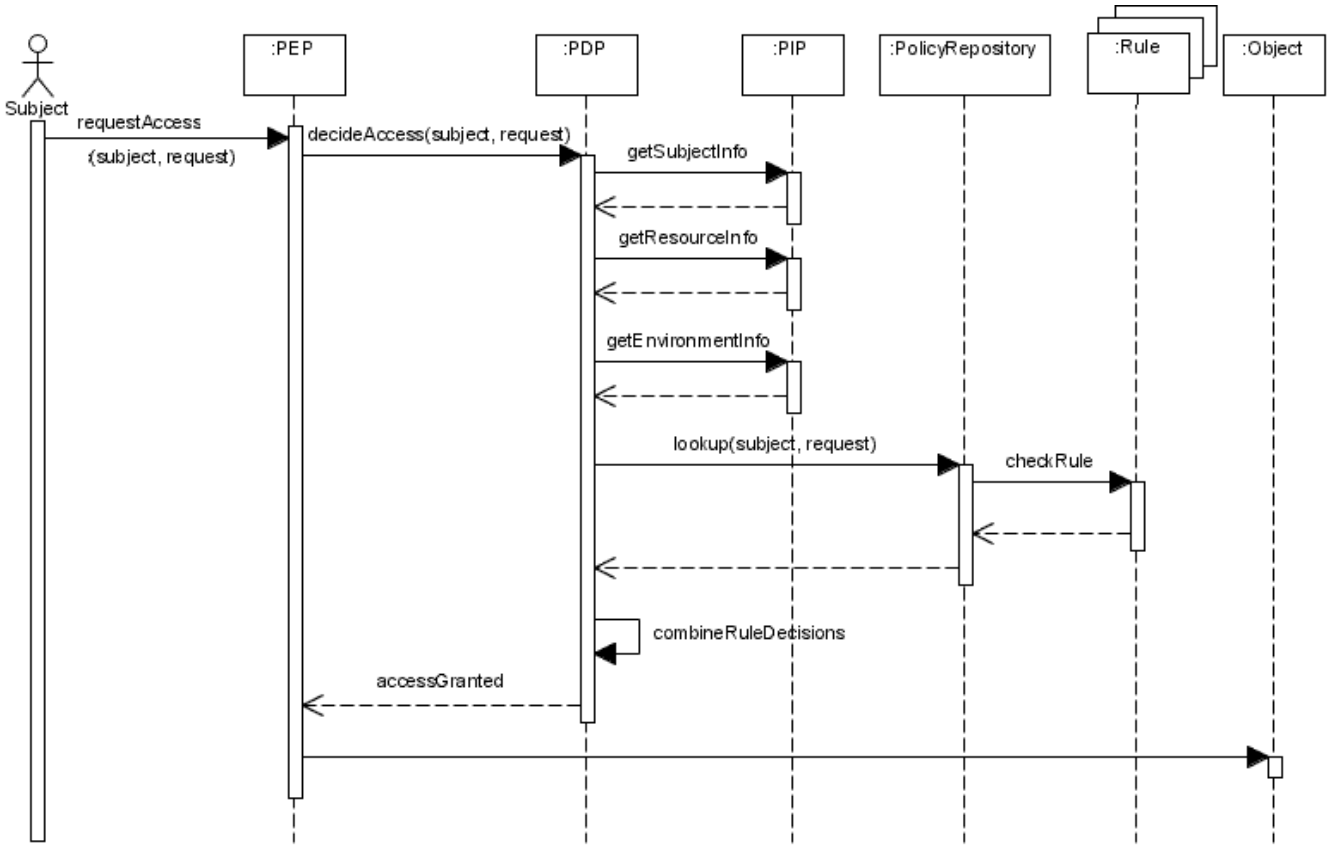

Figure 3. Sequence diagram for use case Request access to an object

Consequences

This pattern presents the following advantages:

- Since the access decisions are requested in a standard format, an access decision becomes independent from its enforcement. A wide variety of enforcement mechanisms could be supported and can evolve separately from the Policy Decision Point.
- This pattern can support the access matrix, RBAC or multilevel models for access control.
- Since every access is mediated, illegal accesses are less likely to be performed.

The pattern also has some (possible) liability:

- It could affect the performance of the protected system since the central PDP/PolicyRepository/PIP subsystem may be a bottleneck in the system.
- Complexity
- We need to protect the access control information.

Related Patterns

XACML patterns [5] is an implementation of this pattern. The Access Control List and the Capability pattern are simple implementations of this pattern.
PEP is just a Reference Monitor [4].
This pattern can implement the Access Matrix and RBAC patterns.

3. ACCESS CONTROL LIST

The Access Control List allows control access to objects by indicating which subjects can access an object and in what way. There is usually an ACL associated with each object.

Example

We are designing a system in which documents should be accessible only to some specific registered users, who can either retrieve them for reading or submit a modified version. We need to verify that a specific user can access the document requested in an efficient manner.

Context

This applies to distributed systems where access to resources must be controlled. Those systems comprise a Policy Decision Point and Policy Enforcement Points that enforce the access policy. A system is composed of subjects that need to access resources to perform tasks. In the system, not every subject can access any object: access rights are defined and can be modeled as an access matrix, in which each row represents a subject and each column represents an object. An entry of the matrix is indexed by a specific subject and a specific object, and lists the types of actions that this subject can execute on this object.

Problem

In some of those systems, the number of subjects and/or objects can be large. In this case, the direct implementation of the matrix

can use significant amounts of storage, and the time used for searching this large matrix can be significant.
In practice, the matrix is sparse. Subjects have rights on few objects and thus most of the entries are empty.
How can we implement the access matrix in a space- and time-efficient way? The solution to this problem is affected by the following forces:

- The matrix may have many subjects and objects. Finding the rule that authorizes a specific request to an object may take a good amount of time (un-ordered entries).
- The matrix can be very sparse and storing it as a matrix would require storing many empty entries, thus wasting space.
- Subjects and objects may be frequently added or removed. Making changes in a matrix representation is inefficient.
- The time spent for accessing a centralized access matrix may result in an additional overhead time.
- A request received by a Policy Enforcement Point indicates the requester identity, the requested object, and the type of access requested. The requester identity, in particular, is controlled by the requester, and may be forged by a malicious user.

Solution

Implement the Access Matrix by associating each object with an Access Control List (ACL) that specifies which actions are allowed on the object, by which authenticated users. Each entry of the list comprises a subject's identifier and a set of rights. Policy Enforcement Points (PEPs) of the system enforce the access policy by requesting the PDP to search the object's ACL for the requesting subject identifier and access type. In order for the system to be secure, the subject's identity must be authenticated prior to its access to any objects. Since the ACLs may be distributed, like the objects they are associated with, several Policy Administration Points (PAPs) may be responsible for creating and modifying the ACLs.

Structure

Figure 4 illustrates the solution. In order to be protected, an **Object** must have an associated **ACL**. This **ACL** is made of **ACLEntrie**s, each of which contains a set of **Right**s permitted for a specific authenticated **Subject**. An authenticated **Subject** accesses an **Object** only if a corresponding **Right** exists in the **Object**'s **ACL**. For security reasons, only the **PDP** can create and modify **ACL**s. At execution time, the **PDP** is responsible for searching an **Object**'s **ACL** for a **Right** in order to make an access decision.

Dynamics

Figure 5 shows a sequence diagram describing the typical use case for Request Object Access. The authenticated **Subject**'s request for accessing an **Object** is intercepted by a **PEP**, which forwards the request to the **PDP**. It can then check that the **ACL** corresponding to the **Object** contains an **ACLEntry** which corresponds to the **Subject** and which holds the **accessType** requested by the Subject.

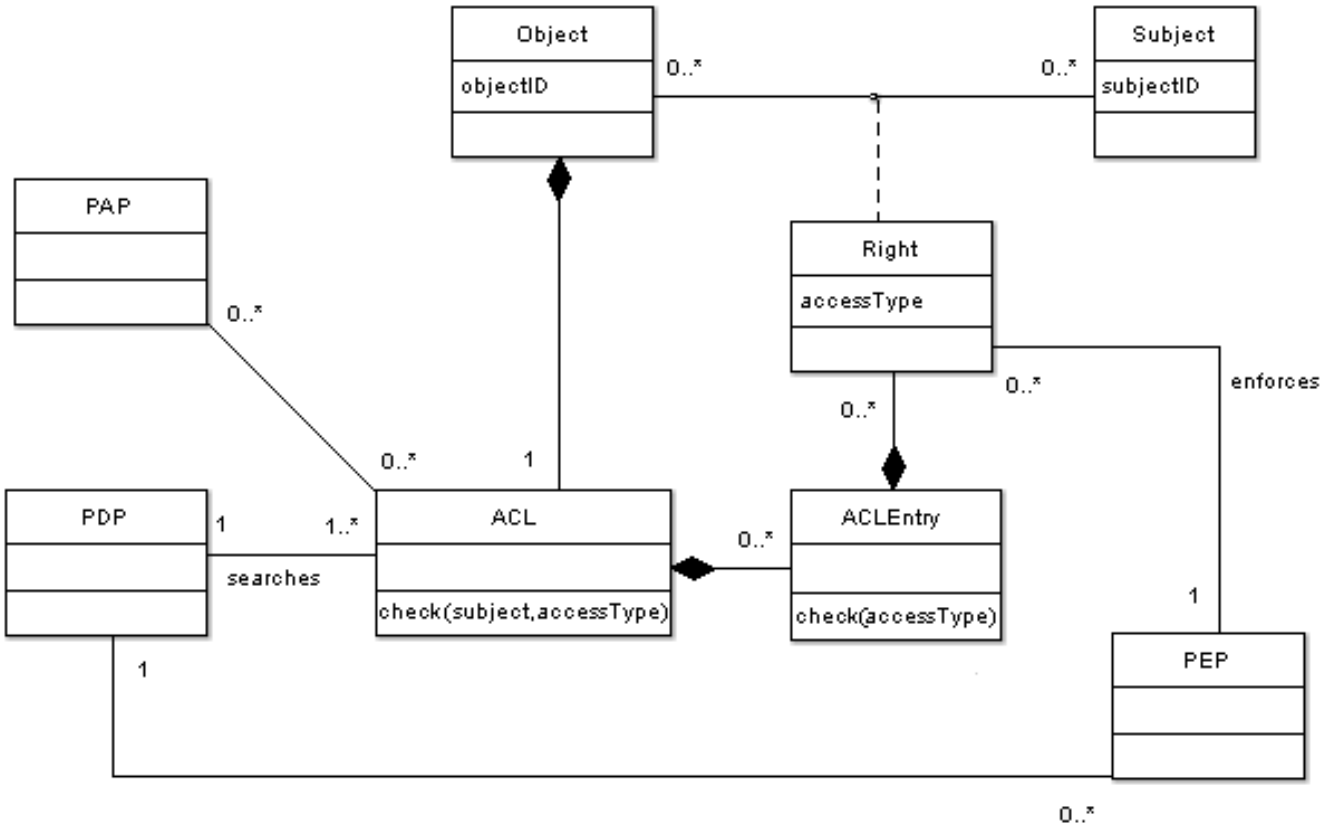

Figure 4. Class diagram for Access Control List

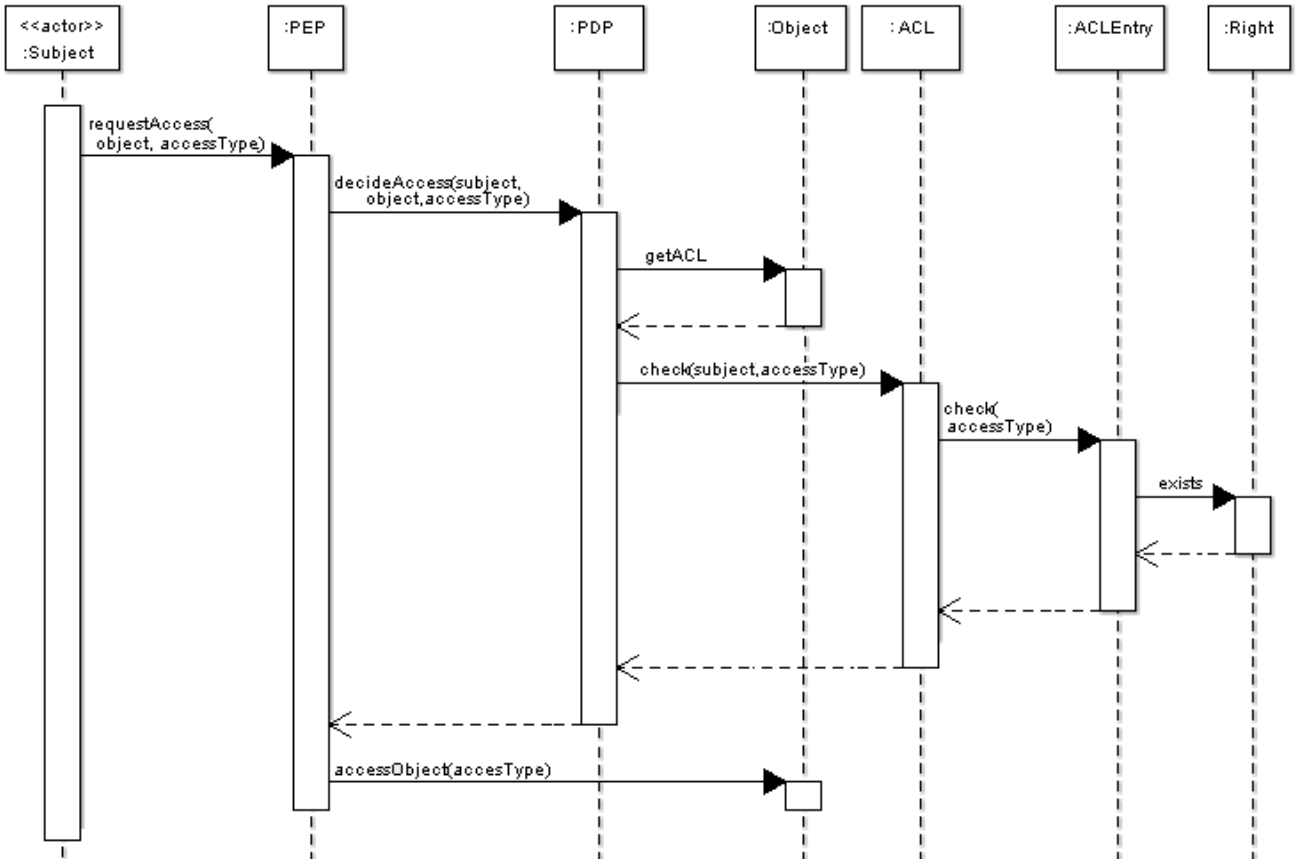

Figure 5. Sequence diagram for use case for Request Object Access

Example Resolved

To enforce access control, we create a Policy Decision Point and its corresponding Policy Enforcement Points, which are responsible for intercepting and controlling accesses to those documents. For each document, provide the Policy Decision Point with a list of the users authorized to access it and in what way (read or write). At access time, the Policy Decision Point is able to search the list for the user. If the user is on the list with the proper access type, it can grant access to the document; otherwise it will refuse access. In our distributed system, we make sure that only authenticated users, that is, users who provided a valid credential, could make requests.

Known Uses

- Operating systems such as Microsoft Windows (from NT/2000), Novell's NetWare, Digital's OpenVMS, and Unix-based systems use ACLs to control access to their resources.
- In Solaris 2.5, file ACLs allow to have a finer control over access to files and directories than the control that was possible with the standard Unix file permissions. It is possible to specify specific users in an ACLEntry. It is possible to modify ACLs for a file 'testfile' by using the 'setfacl' command in a similar way to the 'chmod' command, used for changing standard Unix permissions: setfacl -s u::rwx,g::---,o::---,m:rwx,u:user1:rwx,u:user2:rwx testfile
- IBM Tivoli Access Manager for e-businesses uses ACLs to control access to the Web and application resources [6].
- Cisco IOS Software, Cisco's network infrastructure software, provides basic traffic filtering capabilities with ACLs [7].

Consequences

This pattern presents the following advantages:

- Because all authorizations for a given object are kept together, we can go to the requested object and find out if a subject is there. This is much shorter than searching the whole matrix.
- The time spent accessing an ACL is less than the time that would have been spent accessing a centralized matrix.
- Access to unauthorized objects by subjects submitting forged requests on behalf of legitimate subjects is not possible because we made sure that the requests are from only authenticated subjects.

The pattern also has the (possible) liabilities:

- The administration of the subjects is rendered more difficult: The deletion of a subject may imply the scan of all ACLs, but this can be done automatically.
- When the environment is heterogeneous, it needs to be adapted to each type of PEPs. PDPs and PAPs must be implemented in a different way, thus adding an additional development cost.

Implementation
A decision must be made regarding the granularity of the ACLs. For example, it is possible to regroup the users, such as the minimal access control lists in UNIX.

It is also possible to have a finer-grained access control system. For example, the extended access control lists in UNIX that allow specified access not only for the file's owner and owner's group but also for additional users or groups.

The choice of access types can also contribute to a finer-grained access control system. For example, Windows defines over ten different permissions, whereas Unix-like systems usually define three.

A creation/inheritance policy must also be defined: what should the ACL look like at the creation of an object? From what objects should it inherit its permissions?

ACLs are pieces of information of variable length. A strategy for storing ACLs must be chosen. For example, in the Solaris' UFS file system, each *inod*e has a field called *i_shadow*. If an *inode* has an ACL, this field points to a shadow *inode*. On the file system, shadow *inodes* are used like regular files. Each shadow *inode* stores an ACL in its data blocks. Linux and most other UNIX-like operating systems implement a more general mechanism called Extended Attributes (EAs). Extended attributes are name and value pairs associated permanently with file system objects, similar to the environment variables of a process [8].

Related Patterns
The PEP and PDP come from the previous pattern is this paper. The Capability pattern is another way to implement the Access Matrix.

Access Matrix and RBAC [4] are models that can be implemented using ACLs.

PEP is just a Reference Monitor [4].
A variant with a solution to centralized systems exists; in particular, it leverages on particular data structures to enhance efficiency.

4. CAPABILITY
The Capability pattern allows control access to objects by providing a credential or ticket to be given to a subject for accessing an object in a specific way. Capabilities are given to the principal.

Example
We are designing a system that allows registered users to read or modify confidential documents. We need to verify that a specific user can access a confidential document in an efficient and secure manner. In particular, we worry that if the parts of our system that deal with access control are too large and/or distributed, they may be compromised by attackers.

Context
We refer to distributed systems where access to resources must be controlled. Those systems have a Policy Decision Point and its corresponding Policy Enforcement Points that enforce the access policy. A system is composed of subjects that need to access resources to perform their tasks. In the system, not every subject can access any object: access rights are defined and can be modeled as an access matrix, in which each row represents a subject and each column represents an object. An entry of the matrix is indexed by a specific subject and a specific object, and lists the types of actions that this subject can execute on this object. The system's implementation is vulnerable to threats from attackers that may compromise its components.

Problem
In some of those systems, the number of subjects and/or objects can be large. In this case, the direct implementation of the matrix can use significant amounts of storage, and the time to search this large matrix can be significant.

In practice, the matrix is sparse. Subjects have rights on few objects and thus most of the entries are empty.

How can we implement the access matrix in a space- and time-efficient way? The solution to this problem is affected by the following forces:

- The matrix may have many subjects and objects. Finding the rule that authorizes a specific request to an object may take a good amount of time (un-ordered entries).
- The matrix can be very sparse and storing it as a matrix would require storing many empty entries, thus wasting space.
- Subjects and objects may be frequently added or removed. Making changes in a matrix representation is inefficient.
- The time spent for accessing a centralized access matrix may result in an additional overhead time.
- A request received by a Policy Enforcement Point indicates the requester identity, the requested object, and the type of access requested. The requester identity, in particular, is controlled by the requester, and may be forged by a malicious user.
- The size of the units that can create and/or modify the policies (such as Policy Administration Points) has an impact on the security of the system. Minimizing their size will reduce their chance of being compromised by attackers.

Solution
Implement the Access Matrix by issuing a set of capabilities to each subject. A capability specifies that the subject possessing the capability has a right on a specific object. Policy Enforcement Points and the Policy Decision Point of the system enforce the access policy by checking that the capability presented by the subject at the access time is authentic and searching the capability for the requested object and access type. Trust a minimum part of the system – create a unique capability issuer that is responsible for issuing the capabilities. The capabilities must be implemented in a way that allows the PDP to verify their authenticity, so that a malicious user cannot forge one.

Structure
Figure 6 illustrates the solution. In order to protect the **Objects**, a **CapabilityProvider**, the minimum trusted part of our system, issues a set of **Capabilities** to each **Subject** by using a secure channel. A **Capability** contains a set of **Rights** that the **Subject** can perform on a specific **Object**. A **Subject** accesses an **Object** only if a corresponding **Right** exists in one of the **Subject**'s **Capabilities**. At execution time, the **PDP** is responsible for checking the **Capability**'s authenticity and searching the **Capability** for both the requested **Object** and the requested **accessType** in order to make an access decision.

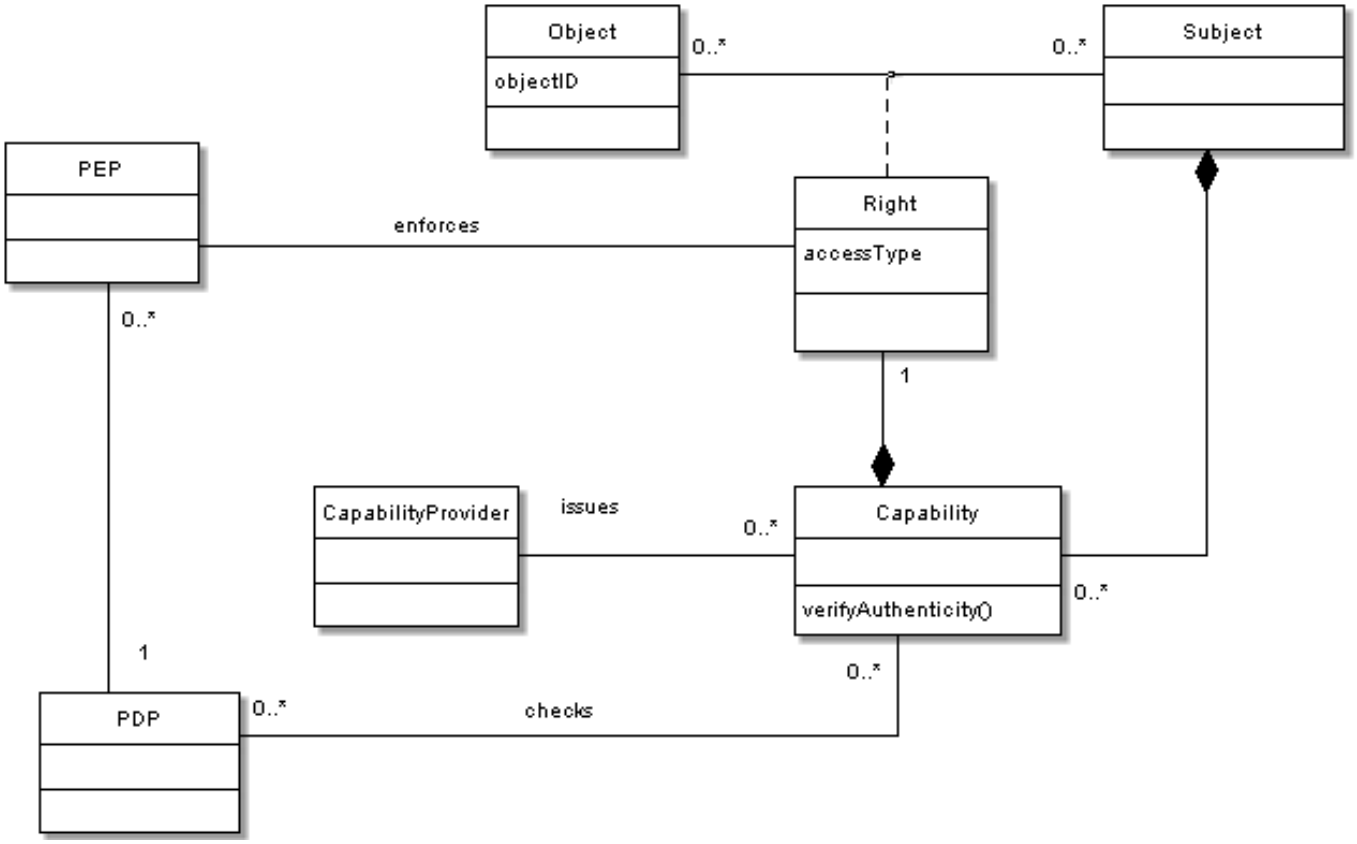

Figure 6. Class diagram for Capability

Dynamics

Figure 7 shows a sequence diagram describing the typical use case of Request Object Access. The **Subject** requests access to an **Object** by including a corresponding **Capability**. The request is intercepted by a **PEP**, which forwards the request to the **PDP**. It can then check that the **Capability** holds the **accessType** requested by the **Subject**.

Example Resolved

To enforce access control, we create a Policy Decision Point and its corresponding Policy Enforcement Points that are responsible for intercepting and controlling accesses to those documents. When a user logs on to the system, a robust token issuer provides a set of tokens that indicate which confidential documents are authorized. Tokens are digitally signed so that they can't be created or modified by users.

At request time, a user wishing to access a confidential document presents its token to the Policy Enforcement Point, and then to the Policy Decision Point, which grants him access to the document. If a user does not present a token corresponding to the document and the access mode, access is refused.

Known Uses

- Most of the capability-based systems are operating systems. Usually hardware assistance is needed, for example, capabilities are placed in special registers and manipulated with special instructions (Plessey P250), or they are stored in tagged areas of memory (IBM 6000).
- Many distributed capability-based systems have been researched and are described in [9, 10, 11, 12]. Among those, Amoeba [12] is a distributed operating system in which multiple machines can be connected together. It has microkernel architecture. All objects in the system are protected using a simple scheme. When an object (representing a resource) is created, the server doing the creation constructs a 128-bit value, called a capability and returns it to the caller. Subsequent operations on the object require the user to send its capability to the server to both specify the object and prove the user has permission to manipulate the object. Capabilities are protected cryptographically to prevent tampering.

Consequences

This pattern presents the following advantages:

- Because the capability is sent together with the request, the time spent for accessing an authorization is much less than the time that would have been spent searching a whole matrix, or searching an ACL.
- The time spent accessing a capability at request time is less than the time that would have been spent accessing a centralized matrix.

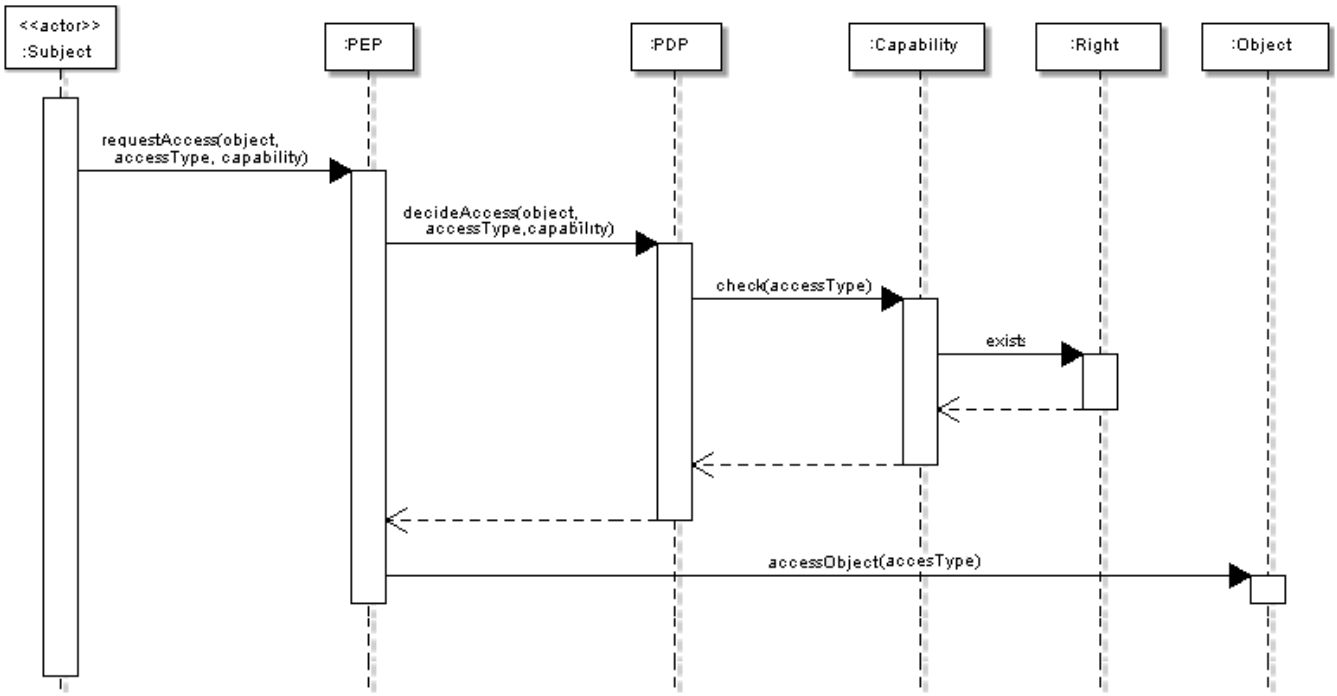

Figure 7. Sequence diagram for use case for Request Object Access

- The part of the system that we need to trust is minimal. The capability provider is only responsible to issue capabilities to the right users at an initial time.
- It is harder for malicious users to forge or modify capabilities, since a capability provides a way to verify its authenticity.

The pattern also has some (possible) liabilities:

- The administration of the objects is more difficult: The addition of an object implies the issuance of capabilities to every authorized user.
- When the environment is heterogeneous, the administration of the rights is more complex. There is no straightforward way to revoke a right since the user is in control of the capabilities it has acquired. A solution could be to add a validity time to each capability, or by through indirection, or by using virtual addresses [13].
- The right is transferable, that is, a capability can be stolen and replayed by (or given to) a malicious user! (This is not the case in OSs in which accesses to the capabilities are also controlled by the TCB, but those need the support of special hardware.)

Implementation

Since a capability must be un-forgeable and un-modifiable, it can be implemented as hardware or software:

- Hardware:
  - Tags: Tagging allows for the categorization of each word as data or a capability. Then no copying should be allowed from capability to data or vice versa, no arithmetic operation should be allowed on capabilities. A disadvantage of this method is the memory waste by using tags.
  - Segmentation: Whole segments of memory are used exclusively for capabilities or for data. No operation should be allowed between partitions of different types. A disadvantage of this is that many processes may need two segments.
- Software: Cryptography is usually used. The capabilities may be encrypted by the capability issuer's key.

Related Patterns

The PEP and PDP are from the previous pattern in this paper. The ACL pattern is another way to implement the Access Matrix.

Capabilities can be implemented into the VAS (Virtual Address Space) using segmentation.

PEP is just a Reference Monitor [4].

Access Matrix, RBAC [4] are models that can be implemented using ACLs. Credentials [14] are a type of capability.

5. CONCLUSIONS

In this paper, we have shown patterns to describe the access control in distributed systems. An abstract one is the Policy-Based Access Control that describes how to decide if a subject is authorized to access an object according to policies defined in a central policy repository. Then we presented implementation-oriented patterns that implement the Access Matrix or RBAC model: The ACL pattern allows control access to objects by indicating which subjects can access an object and in what way. There is usually an ACL associated with each object. The Capability pattern allows control access to objects by providing a credential or ticket to be given to a subject for accessing an object in a specific way. We have also shown the relationships between these patterns and traditional access control patterns for distributed systems.

6. ACKNOWLEDGEMENTS

We thank our shepherd Sami Lehtonen whose valuable comments helped improve this paper. Also, the Secure Software Development Research Group from Florida Atlantic University contributed to improve this paper by providing useful comments and suggestions.

7. REFERENCES

[1] E.B.Fernandez and M.M.Larrondo-Petrie, "A methodology to build secure systems using patterns", *22nd Annual Computer Security Applications Conference (ACSAC), Works in Progress*, Miami Beach, FL, Dec. 11-15. http://www.acsac.org

[2] M. Schumacher, E.B.Fernandez, D. Hybertson, F. Buschmann, and P. Sommerlad, *Security Patterns: Integrating security and systems engineering"*, Wiley 2006.

[3] T. Priebe, E. B. Fernandez, J. I. Mehlau, and G. Pernul, "A pattern system for access control," in *Research Directions in Data and Applications Security XVIII*, C. Farkas and P. Samarati (Eds.), Proc. of the 18th. Annual IFIP WG 11.3 Working Conference on Data and Applications Security, Sitges, Spain, July 25-28, 2004.

[4] E. B. Fernandez and R. Pan, "A Pattern Language for security models", *Proc. of PLoP 2001, The 8th Annual Conference on the Pattern Languages of Programs*, Urbana, IL, USA, 11-15 September 2001. http://jerry.cs.uiuc.edu/~plop/plop2001/accepted_submissions/PLoP2001/ebfernandezandrpan0/PLoP2001_ebfernandezandrpan0_1.pdf.

[5] N. Delessy, and E. B.Fernandez, "Patterns for the eXtensible Access Control Markup Language", in *Proceedings of the 12th Pattern Languages of Programs Conference (PLoP2005)*, Monticello, Illinois, USA, 7-10 September 2005. http://hillside.net/plop/2005/proceedings/PLoP2005_ndelessyandebfernandez0_1.pdf.

[6] IBM Tivoli Access Manager for e-business webpage - http://www-306.ibm.com/software/tivoli/products/access-mgr-e-bus/, Last accessed 2007-06-26.

[7] Cisco IOS Software webpage - http://www.cisco.com/en/US/products/sw/iosswrel/products_ios_cisco_ios_software_category_home.html, Last accessed 2007-06-26.

[8] A. Grünbacher "POSIX Access Control Lists on Linux ", http://www.suse.de/~agruen/acl/linux-acls/online/, Last accessed 2007-06-26.

[9] H.L. Johnson, J.F. Koegel, R. M. Koegel, "A secure distributed capability based system," *Proceedings of the 1985 ACM annual conference on The range of computing: mid-80's perspective: mid-80's perspective*, Pages: 392 – 402.

[10] J.E. Donnelley, "A Distributed Capability Computing System," (DCCS) *Proceedings of the Third International Conference on Computer Communication*, Toronto, Canada, August 3-6, 432-440.

[11] R D Sansom, D P Julin, R F Rashid, "Extending a capability based system into a network environment," *ACM SIGCOMM Computer Communication Review*, Volume 16 , Issue 3 (August 1986) 265 – 274.

[12] Amoeba Operating System's webpage - www.cs.vu.nl/pub/amoeba/, Last accessed 2007-06-26.

[13] E. B. Fernandez, T. Sorgente, and M. M. Larrondo-Petrie, "Even more patterns for secure operating systems," *Procs. of the Conference on Pattern Languages of Programs, PLoP 2006*, Portland, OR, October 2006, *http://hillside.net/plop/2006/Papers/Library/Even_more_patterns.pdf*

[14] P. Morrison and E.B. Fernandez, "The Credential Pattern," *Procs. of the Conference on Pattern Languages of Programs, PLoP 2006*, Portland, OR, October 2006, *http://hillside.net/plop/2006/Papers/Library/PLoP2006_Credential.pdf*