R | R | Z | N

Leibniz Universität Hannover

Gottfried Wilhelm Leibniz Universität Hannover
Regionales Rechenzentrum für Niedersachsen
Fachgebiet Distributed Virtual Reality
Lehrgebiet Rechnernetze

Masterarbeit
im Studiengang Informatik (M. Sc.)

Policy-based Authorization for Grid Data-Management

Verfasser:	B. Sc. R. Gröper
Erstprüfer:	Prof. Dr.-Ing. C. Grimm
Zweitprüferin:	Prof. Dr.-Ing. G. v. Voigt
Betreuer:	Dipl.-Ing. S. Piger
Datum:	13. Oktober 2006

Hannover, den 13.10.2006
Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ralf Gröper

Abstract

Grid computing is a new paradigm for accessing resources distributed across organizational boundaries using computer networks. It allows for new ways of accessing and using remote resources. This implies a demand for several new concepts concerning security, namely authorization and authentication. The most common security framework used in today's grid implementations is the Grid Security Infrastructure. It makes use of proxy certificates and private keys as credentials for allowing single sign on and delegation of rights. These credentials can, if intercepted, be used to hijack the identity of a user and, for example, access her data. In this thesis I will explore the current state of data security and investigate methods for improving security within data management. One approach will be presented by a working prototype implementation of user based data access policies embedded inside proxy certificates.

Contents

List of Figures

List of Abbreviations

ACL	Access Control List
API	Application Programming Interface
CA	Certificate Authority
CAS	Community Authorization Service
CDT	Eclipse C/C++ Development Toolkit
$CERN$	Organisation Européenne pour la Recherche Nucléaire
CN	Common Name
CRL	Certificate Revocation List
CVS	Concurrent Versions System
$DBMS$	Database Management System
DOM	Document Object Model
EEC	End Entity Certificate
$EGEE$	Enabling Grids for E-SciencE
FAS	File Authorization Service
$FiReMan$	File and Replica Manager
FTS	File Transfer Service
gnu	Gnu's Not Unix
$GridFTP$	Grid File Transfer Protocol
GSI	Grid Security Infrastructure
$GT4$	Globus Toolkit 4

$GUID$	Globally Unique Identifier
$HTTP$	Hypertext Transfer Protocol
IDE	Integrated Development Environment
$IOService$	Input/Output Service
LFN	Logical File Name
$OCSP$	Online Certificate Status Protocol
OID	Object Identifier
PC	Proxy Certificate
PDP	Policy Decision Point
PEP	Policy Enforcement Point
PKI	public key Infrastructure
RPM	Red Hat Package Manager
$RRZN$	Regionales Rechenzentrum für Niedersachsen
$SAML$	Security Assertion Markup Language
SRM	Storage Resource Manager
SSO	Single Sign-On
$SURL$	Site Uniform Resource Locator
TCP	Transmission Control Protocol
$TURL$	Transport Uniform Resource Locator
VO	Virtual Organization
$VOMS$	Virtual Organization Membership Service
WMS	Workload Management System
$XACML$	eXtensible Access Control Markup Language

Chapter 1

Introduction

1.1 About Grid Computing

The *Grid* will be, as many of its advocates think, the next evolutionary (or even revolutionary?) step in organizing networked electronic resources. Ubiquitous access to these resources, including but not limited to computing resources, mass storage, knowledge bases and sensors, will transform our society and the way we think about and use computing [FoKe04].

The problem that led to the development of Grid technologies occurred first in scientific communities which needed access to computing and storage resources exceeding the locally available resources. The idea to combine resources owned by different sites of that community to one large virtual computer led to a definition of a computational Grid. This Grid is designed to offer features known from the power grid: Access to arbitrary networked resources shall be available from the network outlet in the wall just like electrical power from the power outlet. The user of the Grid shall not be concerned with the exact location of these services, i.e. whether they are provided by the local data processing center or by a remote one. The Grid infrastructure will accept jobs from the user, find a suitable resource to handle the request and return the results to the user.

This automatic distribution of jobs is the fundamentally new concept in Grid computing: In classic environments jobs could only be submitted to batch systems feeding the job to one local computing resource, e.g. a computing cluster. If the local resource is busy the user might have to wait for a long time and if the resource is either to small or has a unsuited architecture the user might not be able to get a result at all. By allowing job submission without specifying an exact target resource but by

specifying the requirements of the job this problem can be overcome when there is a non-local resource available meeting these requirements. A community of users and resource providers sharing their resources in such a way is called a *Virtual Organization* (VO).

The advantages for the user, as explained above, are manifold. The need for this new technique or set of different techniques working closely together has however not yet reached the end customer. Main actors in the field of Grid computing are universities and research departments of computer industries. One example where Grid technologies are being used to solve a scientific problem is the EGEE Project with is leading the development of the gLite Grid middleware. The EGEE project is driven by the need for a large scalable infrastructure for processing data produced by the Large Hadron Collider currently being built at CERN in Switzerland. Using Grid technologies will enable researchers all over the world to access results of experiments and do computations based on these results, even if their local site does not provide sufficient resources.

1.2 About this Thesis

In a Grid environment new and additional security and privacy requirements over classic local job handling are imposed. The extent of these requirements differs from community to community. One community where non-sensitive results of experiments are shared among all members of the VO there is no need for fine-grained resource access authorization whereas other communities have strict privacy requirements imposed by law, e.g. for medical communities, or to prevent industrial or scientific espionage in highly competitive communities.

This thesis is concerned with evaluating security mechanisms currently used by Grid middlewares, specifically those using the Grid Security Infrastructure, and finding a pragmatic way to improve security by identifying weaknesses and evaluating solutions. This is done by explaining and evaluating the GSI in the second chapter.

As special focus in this this thesis is laid on data management and access, in chapter 3 general data management techniques in Grids are explored and specific implementations in the gLite middleware and the Globus Toolkit are compared.

A way of minimizing potential damage done by exploiting the security weaknesses discovered in chapters 1 and 2 is presented in chapter 4: User self-restriction of

rights in form of User-Based Policies (UBP). The concept is explained and possible implementations for different Grid architectures are discussed.

In chapter 5 the gLite infrastructure available at the RRZN is enhanced by a working prototype implementation of the aforementioned concepts: First, a XACML-based data policy format is defined to express User-Based Policies. Second, the voms-proxy-init tool used by gLite to create proxy certificates is enhanced to allow for the inclusion of such UBPs. Third, the IO Service policing file access is enhanced to interpret and enforce UBPs.

Finally, a summary of the thesis is given, a conclusion drawn and an outlook for future work presented.

Altogether, the current state of policing data access in Grid middlewares using the Grid Security Infrastructure is analyzed, weaknesses and possible damage done by exploiting these weaknesses is identified and solutions for minimizing that damage are presented. Furthermore the solution for a specific architecture, i.e. a gLite-based Grid using the IO Service, is implemented and presented.

Chapter 2

Security in Grid Computing

In this chapter security mechanisms used by Grid middlewares relying on the Grid Security Infrastructure (GSI) are presented. First, I will explain mechanisms used by the GSI, such as X.509 certificates. Thereafter requirements for Grid security such as Single Sign-On and Delegation of Rights will be discussed and by what means the GSI implements these requirements.

The foundation in Grid authorization techniques laid out in this chapter is needed to understand the authorization mechanisms presented in chapter two. Furthermore the security concepts explained in this chapter will be used by the User-Based Policies for user self-restriction of access rights developed in chapter 4 and their implementation presented in chapter 5.

2.1 General Security Concepts

Security in IT-Systems is not a state but a process which needs constant evaluation and readjustment. Computer security is composed out of *availability*, *integrity* and *confidentiality* of data and resources. This thesis is concerned with confidentiality only. Confidentiality means that access to resources and data is only made available to legitimate users or entities and by no ways illegitimate users or entities may access these resources or data. The most relevant concepts for ensuring confidentiality are:

- *Authentication*: The action necessary to confirm that an entity is in fact the entity it pretends to be. This is done by evaluating some kind of credentials which identify the requesting entity and which cannot be used by (e.g. bio-

metric information) or are not known to (e.g. passwords or private keys) other entities.

- *Authorization*: The action necessary to confirm that an authenticated entity is allowed to perform an action on a resource. This is governed by some policy information available to the authorizing entity.

There are many other concepts and requirements which are important for creating secure IT-Systems but these are out of the scope of this thesis.

2.2 public key Infrastructures

A public key Infrastructure (PKI) allows for secure authentication of entities in computer environments, especially in networked environments. PKI uses asymmetric cryptography to bind the public key of a keypair to entities by signing it together with meta information about its owner. The main advantage over concepts not using asymmetric cryptography is the fact that there is no need for shared secrets which would have to be exchanged over a potentially unsecure network or by other means to establish an authenticated or encrypted connection.

2.2.1 X.509 Certificates

A public key annotated with meta information and signed by a third party is called a *certificate*. The most common framework for implementing a PKI is ITU-T Recommendation X.509 [ITU05], thus all certificates discussed in this thesis are X.509 certificates. As the ITU-T Recommendation X.509 is only a framework for building PKIs the concrete implementation is defined in RFC 2459 [HPFS02].

Every private key can be used to sign certificates, but typically only certificates signed by a trusted Certificate Authority's (CA) private key will be accepted. The CA is the root of trust in a PKI which all resources validating e.g. signed emails must trust to the extent that certificates signed by that particular CA are indeed representing the person or entity being stated by the certificate's meta information.

The CA's root certificate must be available on all resources that need to validate certificates derived from it. The CA's root certificate is not secret and thus does not have to be encrypted for transfer, but nevertheless it must be well protected against

unauthorized write access so that it cannot be exchanged by a forged one allowing an attacker to create and sign certificates which in turn would be accepted as valid.

Certificates which are not root certificates of a CA but belong to some entity are called End-Entity Certificates (EEC). The End-Entity can be a natural person or another entity like a computer or a service. The integrity of EECs can be validated by cryptographically verifying the signature of the EEC in question with the CA's public key taken from the root certificate of that CA.

As this concept is based upon asymmetric cryptography, every certificate also has a private key belonging to the keypair consisting of this private key and the public key from the certificate. The private key must be kept secret at all times as anyone in possession of it may act in the name of the entity stated in the meta information within the certificate. Sending a private key over a network is thus not desirable. The key is then out of control of the owner and she can not guarantee that it is kept save at all times. It could be transferred over unsecure networks or it could be exposed inside a file system on a resource. As EECs have a default validity time of several months to years and CA root certificates of tens of years, hijacked private keys could be used for the whole rest of the lifetime of this certificate if the owner does not observe the misuse and asks the CA for revocation of the certificate.

Certificate revocation is not reliable as the *Certificate Revocation Lists* (CRL) containing information about revoked certificates have to be transferred to all nodes by hand or cron-jobs and will thus not always be up to date. Even protocols allowing for real time certificate validation like the Online Certificate Status Protocol (OCSP) defined in [MAMG99+] do not provide complete security as the owner might not notice that the private key has been compromised for some time, e.g. until she is billed for the actions of the attacker. In addition OCSP is not yet implemented on a wide basis.

2.2.2 X.509 Certificate Extensions

Certificate extensions are a mechanism to embed additional information within X.509 certificates introduced in X.509 version 3 (X.509 v3). Several standard extensions, e.g. for specifying a pointer to the certificate belonging to the private key used to sign the current certificate, are defined in RFC 2459. It is also possible to define additional extensions for embedding arbitrary information inside a certificate which will be signed by the CA together with the rest of the certificate, thus rendering this information unforgeable.

In order to distinguish different extensions, they must be identified by a unique *Object Identifier* (OID). OIDs are organized in a hierarchical namespace and defined in ITU-T recommendation X.680 [ITU02]. After being assigned a base OID it is possible to create new OIDs for different purposes by appending new leaves to the branch of the tree defined by this assigned OID. Every institution with an own base OID can thus create an unlimited number of OIDs for arbitrary purposes. One of these purposes is the unique identification of certificate extensions.

Extensions must be marked as *critical* or *non-critical*. Critical extensions must be interpreted by all systems using the certificate in some way. If a system does not know about a critical extension, it must reject the whole certificate. Non-critical extensions in contrast do not have to be be interpreted by all systems, thus systems may accept a certificate as valid even if it contains an unknown non-critical extension.

2.3 The Grid Security Infrastructure

The Grid Security Infrastructure (GSI) is the prevailing implementation for security functionality in today's major Grid middlewares. Both the Globus Toolkit[1] and gLite[2], which is partially based upon Globus, use the GSI. It has been introduced first with the Globus Toolkit version 2 and is since then constantly enhanced. Especially porting the GSI to a web service based architecture is an aim of current developments [FoKe04]. Authentication using the GSI is based upon a public key Infrastructure using X.509 proxy certificates described later in this chapter.

2.3.1 Delegation of Rights and Single Sign-On

Delegation of Rights and Single Sign-On are the driving requirements for any security framework for Grid environments. These two basic requirements have to be met so that Grid computing as it is envisioned today and explained in chapter 1 will work.

Single Sign-On (SSO) is a concept allowing users to gain access to multiple systems they have the rights to access by entering their credentials, e.g. a username/ password combination, only once. SSO is needed by Grid environments because potentially long running jobs which pass through and access multiple systems at multiple sites within the VO must be able to do so without inquiring the user each time. This needs

[1] http://www.globus.org
[2] htt://www.glite.org

a certain level of trust between those systems. When the user is authenticated to the Grid, all participating resources must trust in that this authentication is valid.

Delegation of Rights is needed to enable the Grid components to act on behalf of the user with the rights of the user. Other resources within the Grid must be able to e.g. fetch some files which the user has read access to. The computer that particular service is running on might not have the rights to do that itself and, if the data is stored at a different site under different administration, it would increase the administrative overhead even more. Of course such a procedure would not be practical or desirable anyway because there would not be an easy way to have user- or role-based access rights within one VO .

Both, Single Sign-On and delegation of rights, are tightly coupled in the Grid Security Infrastructure and they can not be viewed individually. When a service requests something from another service on behalf of the user both concepts are needed as a whole: The user might not be at her desk to answer a password inquiry, thus needing Single Sign-On, and the requesting service has to be given the user's access rights to the requested resource, thus needing delegation of rights. Both functionalities are provided by issuing proxy certificates explained later in this section.

2.3.2 Mutual Authentication

Not only users but also services and hosts have to authenticate each other to ensure that a connection really comes from another peer itself and not from an attacker pretending to be that peer. This is achieved by using the GSI as well: Both partners need certificates signed by CAs both parties trust. The following steps ensure a mutual authenticated connection initiated by A:

1. Peer A sends peer B its certificate.
2. B checks the validity of the certificate with the trusted CA's certificate.
3. If the certificate is found to be valid, B has to make sure that A really is A and does not only present A's certificate. To achieve this B generates a random message and sends it to A. A now encrypts this message using it's own private key and returns the encrypted message to B
4. B decrypts the message using A's public key taken from the certificate. If the decrypted message matches the original message, B knows that A really is A.
5. Now B sends A it's certificate and the process repeats the other way round.

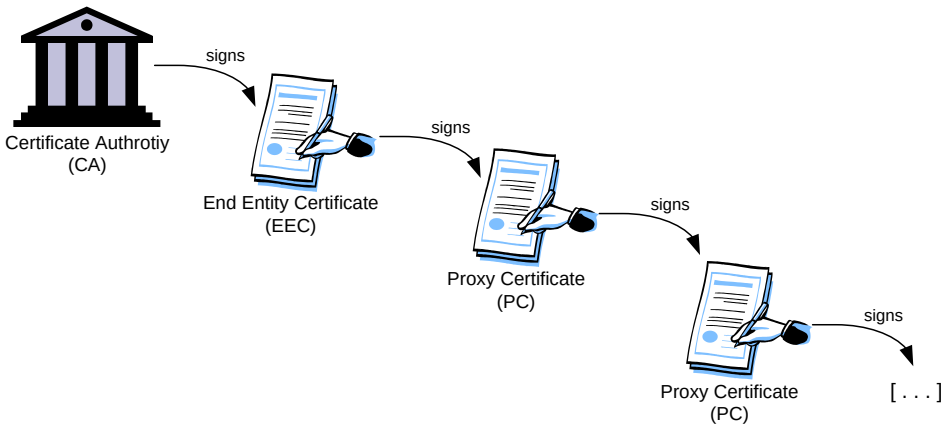

Figure 2.1: Chain of Trust with proxy certificates

After this procedure both peers have a mutually authenticated connection. This connection is not automatically encrypted but it is an easy task to securely exchange a symmetric session key using the certificate's public keys for encryption.

If proxy certificates, as described in the next subsection, are used, the above process becomes slightly more complex. In this case it does not suffice to transfer the proxy certificate to the peer as it is not signed by a trusted CA but by the user herself. To keep the chain of trust intact the proxy certificate and the EEC have to be exchanged. Now the validity of the user's EEC can be validated using the CA certificate. If the EEC is accepted as valid, it can then in turn be used to validate the proxy certificate. The chain can have arbitrarily many elements by creating PCs derived from other PCs, but every single (Proxy-) certificate starting with the EEC has to be included into the list of certificates sent to the peer.

2.3.3 X.509 Proxy Certificates

The GSI allows for Single Sign-On and Delegation of Rights by using a combination of the user's certificate and the associated private key as *credentials*. These credentials are sent to the network in unencrypted form as Grid components need to access them. As the private key is a secret information only known to the user this form of credentials enables Grid components to act in the user's name.

As described earlier in this chapter it is however not favorable to let a private key out of control of the owner. This problem cannot be completely avoided without changing the whole concept of the GSI. Thus a solution has been chosen that considerably

lessens the possible damage done by abused credentials: Users act as their own CAs by creating so called *proxy certificates* (PC) derived from and signed by their long running EEC and the corresponding private key respectively. A proxy certificate has a short lifetime, typically 12 hours, and is used in conjunction with its private key for delegating the user's rights to the Grid. It is also possible to derive PCs from other PCs, but they can only have a maximum lifetime until the end of the issuing certificate's lifetime.

The user's EEC has also to be included into the credentials to keep the chain of trust starting at the CA's certificate (see fig 2.1) intact, but the user's private key can be kept secret on the user's computer, a key card or an online credential store like MyProxy [MyPr06], which is part of both GT4 and gLite 3.0.

Additionally, to further limit the number of possible attacks, it is possible to avoid sending private keys over the network at all. This is done in GSI by creating a new keypair on each resource receiving a delegation. The public key of this keypair is sent back to the delegating resource which in turn creates a certificate containing the information from the original proxy certificate and signs it with the private key of that original PC. The original and the newly created PCs are forwarded to the resource receiving the delegation, but the original private key is kept secret on the delegating resource. Now the resource which received the delegation has got the user's credentials in form of a certificate chain (which now has one more element) and the private key of the last PC in that chain and no private keys have been transferred over the network.

Proxy certificates in gLite can be identified by the subject field. It must be identical to the subject of the issuing certificate but extended by one more Common Name (CN) field containing the word "proxy". As it is also possible to derive PCs from other PCs, the number of CNs with the value "proxy" indicates how long the certificate chain from the EEC to the current PC is.

An attempt to standardize proxy certificates is documented in RFC 3820 [TWEP04+]. At this time RFC 3820 compliant PCs are only supported by GT4[3], gLite does not currently support them. RFC 3820 compliant PCs do not use CNs containing "proxy" but an ProxyCertInfo extension tagging a certificate as a proxy certificate. The use of an extension instead of CNs allows for the inclusion of additional information for defining or limiting the use of the PC. RFC 3820 compliant PCs allow for exam-

[3]GT4 does not yet use the OID defined in RFC 3829 to identify the *ProxyCertInfo* extension but a proprietary one. This has accidentally been released in the final version and will be corrected in the future. [Wel05]

ple to limit or deny the derivation of other PCs from the PC. Another feature is the inclusion of policies inside the PCs in a standardized way.

2.4 Authorization Techniques

2.4.1 Policy Decision Points and Policy Enforcement Points

For effective authorization two distinct actions must be performed: A authorization request must be decided upon and the result of the decision must be enforced. These two steps do not necessarily have to be carried out by the same entity but can be distributed between distinct entities, subsequently called *Policy Decision Points* (PDP) and *Policy Enforcement Points* (PEP).

Policy Decision Points

A PDP is a logical entity that makes authorization decisions. For making an authorization decision three kinds of information, as described in [FVCG00+] for a generic Authentication, Authorization, and Accounting architecture, are required:

- The resource to be accessed and the desired method of access, e.g. read a or write to a file.
- Information about the requester, e.g. her user name and/or group memberships.
- The operating environment, i.e. authorization information. This information is also called a *policy*.

It is possible that more than one rule from the policy applies to a requested access by a specific requester. In this case a *rule combining algorithm* has to be executed. Three possible algorithms are:

1. *Deny-Overrides*: If at least one rule denies the requested access, access is denied regardless whether there are other rules granting access.
2. *Allow-Overrides*: If at least one rule grants access, access is granted regardless whether there are other rules denying access.
3. *First-Applicable*: The first rule found by searching the available authorization information is being followed. Possible further rules are not evaluated at all.

Which of these algorithms to use depends on the actual environment that the authorization decision is made in. For instance in an environment where it is not possible to have contradictory policies using First-Applicable will increase performance as not all rules have to be evaluated but only the rules until the first matching rule is found.

Policy Enforcement Points

A PEP is the logical entity which enforces a decision made by a PDP, i.e. it technically realizes the grant or denial of a request. The PEP does not make its own authorization decisions. There are several ways for the interaction between the PDP and the PEP as described in the next section.

2.4.2 Centralized vs. Distributed Models

In authorization frameworks the authorization information itself (i.e. the policies) or the PDPs can be distributed between more than one system or they can be managed by a central entity. Generally, centralized models are easier to administrate and making policy decisions is easier but the central server can be a single point of failure or a bottleneck limiting the performance of the whole system. A distributed model is in contrast more difficult to administer and PDPs have to make sure that all relevant policies are considered for making an authorization decision. On the other hand distributed models are generally more scalable.

2.4.3 Agent, Push and Pull Models

The rules governing which authorization requests to grant and which ones to deny can be acquired by the PDP in different ways [VCFG00+].

The *agent* model (fig. 2.2A) lets the requesting entity send its request not to the resource to be accessed directly but to a third server (i.e. the PDP) which in turn has to have access to the relevant policy information. If access is granted this server forwards the request to the resource, if not it presents a rejection message to the requester. In this case the PEP is located on the authorization server. If the access is granted, the usage of the resource can be relayed through the agent but it is also possible to let the requester and the resource communicate directly after the access has been granted. If one agent is responsible for a great number of requesters and

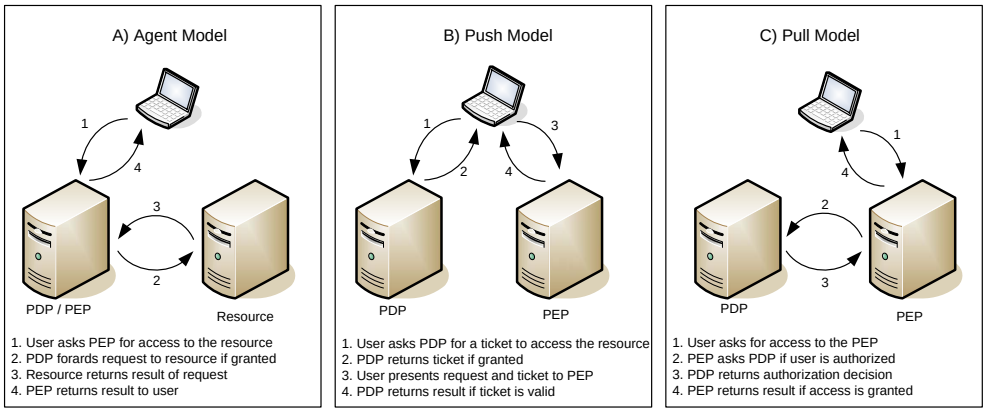

Figure 2.2: Agent, Push and Pull Models

resources this might become a bottleneck limiting the performance of the whole system.

In a *push* model (fig. 2.2B), the requesting entity directs its request directly to the resource to be accessed. The PDP within this resource gets the authorization information without querying it but it will be supplied together with the request. The requesting entity collects the necessary information by acquiring some sort of ticket from an authorization server (i.e. the PDP). The requester presents this ticket together with the access request to the resource. The PEP is then located on the resource. As the requester supplies this information it has to be cryptographically signed or made unforgeable by other means so that the PDP can be sure that the information is originating from the responsible authorization server. The ticket can have a limited lifetime to restrict it to immediate use.

The *pull* model (fig. 2.2C) leaves the responsibility for gathering relevant authorization information to the PDP. After a request has been received the PEP has to pull authorization information actively, e.g. from a authorization server (i.e. the PDP). Like in the push model the PEP is located on the resource.

2.5 Security Weaknesses

This section describes security problems that arise from the Grid Security Infrastructure. It is only concerned with systematical problems caused by the properties and the design of GSI. This thesis is not concerned with problems which might arise from programming bugs in current implementations or attacks based on social engineering, e.g. guessing of passwords. It also does not cover issues caused outside of the

GSI like exposure of private keys of EECs on the user's system or in online credential repositories, and the use of too short keys or insecure algorithms for asymmetric cryptography.

The main security weakness of the GSI derives from abuse of stolen proxy certificates. The user has no way of verifying that her credentials are not hijacked by some attacker. The job will not even fail if someone takes a copy of the user's credentials. Furthermore there is no revocation mechanism for proxy certificates available, so even if the user notices that her credentials are abused she cannot revoke them. This problem is not completely solved by the limited lifetime of proxy certificates. For the time of the validity of the proxy certificate the attacker can act in the name of the user and exploit all her rights. If there is an automatic proxy renewal mechanism, e.g. using the MyProxy credential management service part of both gLite and Globus, this time can even be extended by the attacker.

Possible points of attack within the Grid are:

- The attacker has root access to a resource within the Grid. The attacker might be a local administrator of the resource or the resource might be compromised by other means. As credentials are stored in unencrypted form on the local file system, e.g. in the temporary directory, the attacker has access to all credentials delegated to that resource.
- The credentials are sent over an unencrypted network the attacker has access to. As explained above, private keys are not normally sent over the network but a new keypair is instead created on the receiving resource, but if RFC3920 compliant PCs with restricted derivation of further PCs are used it will become necessary to transfer complete credentials including the private key.
- In larger VOs Grid users are not mapped to individual local accounts but to *Pool Accounts* on the resources their jobs are executed on. This means that a Grid user will be mapped to a local user account that another Grid user might have used shortly before. If such a Pool Account is not carefully cleaned up after use including the temporary credentials, the following user might be able to read them from the temporary directory as she will have the same local user ID.

The use of proxy certificates instead of EECs limits only the lifetime of the credentials but within this lifetime the attacker can act in the name of the user or resource

whose credentials he acquired. The scope of rights of the credentials including the proxy certificate is identical to the full scope of rights the user has.

As it is impossible to guarantee that all resources and all administrators, both on VO and site local level, within the Grid are trustworthy this security weakness cannot be eliminated without completely turning away from using the GSI. Furthermore the Grid is composed of a multitude of heterogeneous systems and resources and thus it will not be possible to guarantee that all these systems are secure in a way that it is impossible for attackers to obtain credentials of the Grid's users. The only way to enhance security in a sustainable way is by finding a pragmatic solution to minimize the potential damage done by abused hijacked proxy certificates.

Chapter 3

Grid Data-Management

In this chapter data-management in Grid environments will be explained. The main service types and their interaction will be discussed. Thereafter implementations of these services in current Grid middlewares, namely gLite 3.0 and the Globus Toolkit 4.0, are examined.

This overview over Grid data-management is needed to understand the design decisions made in chapter 4 for the general concept of user self-restriction of rights by issuing User-Based Policies. Furthermore, in chapter 5, gLite's IO Service presented in this chapter will be extended to support UBPs.

3.1 Basic Principles of Grid Data-Management

In grid environments data-management services are among the key components besides execution management, logging and bookkeeping services and so on. Execution jobs may need input data on the computing element and the output has to be transferred to the user or to a remote storage element. Furthermore, data from a central source might be replicated to multiple locations to provide efficient access at different sites. Concepts used by local sites for authorization like standard UNIX file access control by local user IDs and group IDs are not easily transferable to Grids as VOs are by their nature distributed and dynamic and would thus require a complex and error-prone synchronization technique between these sites.

Additionally site administrators do seldom agree to let local user control out of their hands and delegate that task to VOs. New concepts and mechanisms are thus needed to enable secure, efficient and reliable data-management in Grids.

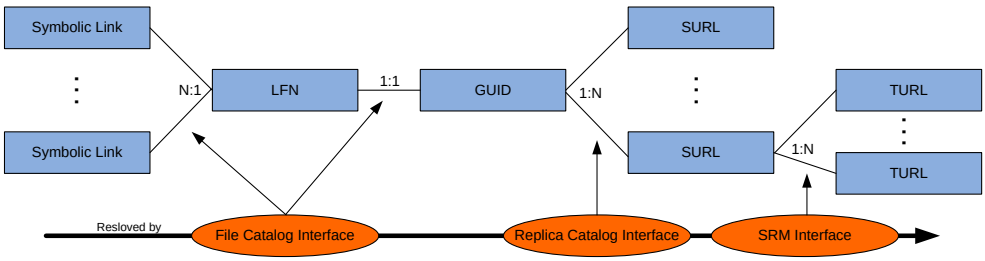

Figure 3.1: Resolving of Filenames to Transports URLs

In order to develop Grid data-management infrastructures first of all the possible types of data resources have to be identified. The most common are:

- Flat files on different storage devices ranging from simple hard disks to sophisticated robotic tape servers.
- Data streams can be produced e.g. by sensors.
- Structured Data may be located in Database Management Systems (DBMS).
- Catalogs include meta data about data available in the Grid.

Both, gLite and the Globus Toolkit, support only access to flat files and catalogs for storing meta data as today's major communities using Grid technology mainly work with data stored in flat files. Database systems and contiguous data streams are not directly supported but can be made available by a suitable abstraction layer.

Users who want to access any of the above resources must be authenticated and authorized to do so. The simplest approach is to base access rights on VO membership, i.e. every user authenticated to be member of a VO is able to access all files that are available in the VO. As it should be possible to define access permissions an a per user basis to ensure data privacy and confidentiality on a per user basis this approach is too coarse. Local user access permissions on the storage systems are, as there might not be a one-to-one mapping of Grid identities to local identities, not a viable solution. Thus a file authorization service (FAS) storing access policies on a per user basis is needed. Such a FAS will, if queried with a desired data access and the identity of the requester, grant or deny the access in question.

3.1.1 Representations of Files

In Grids based on the gLite middleware files can be referenced in multiple ways, depending on the abstraction layer on which the file is referenced. The possible types of references are:

- *Symbolic Links* are, analogous to symbolic links on UNIX file systems, links to other files that have no own physical representation of the file but they are just pointers to that file. Symbolic links are thus a way to make a file accessible from multiple locations within a directory tree without consuming additional space.
- A *Logical Filenames* (LFN) is the human readable representation of a file in the Grid. In principle an LFN can be an arbitrary string, but they simulate a hierarchical file system by interpreting the character "/" as directory delimiters. This way two technically unrelated files can be placed in the same virtual directory if their LFNs only differ in the part after the last "/". LFNs are mutable by the user, i.e. a user can rename the LFN of a file.
- The *Globally Unique Identifier* (GUID) of a file is the unique immutable identifier for a file in the Grid. It is used to unambiguously reference a file by the Grid components. It cannot be changed by the user and should not be exposed to the user in normal operation.
- A *Storage URL* (SURL) defines a specific location of a file. As a file referenced by a GUID can be replicated at multiple sites a SURL defines which Storage Resource Manager is responsible for accessing the file.
- A *Transport URL* (TURL) specifies the exact location and transfer protocol by which a file is accessible. The transfer protocol could be e.g. GridFTP and the location of the file the hostname of the Storage Element and the local path of the file. As a file might be accessible using different protocols more than one TURL might be associated with a SURL. The TURL does not have to be static but can be temporarily assigned for a single access to a file.

The interrelationship between those file reference types and their cardinality can be seen in figure 3.1. Also can be seen which services described in the next sections are responsible for resolving one type into another, where typically only resolving from left to right is possible, i.e. it is not possible to (easily) find out what the LFN for a given TURL is.

3.2 Metadata Management Services

The File, Metadata and Replica Catalogs are basically databases with a networked query interface. It is possible to combine these catalogs in any combination into one service.

3.2.1 File Catalogs

The File Catalog manages the mapping between LFNs and GUIDs. Furthermore it may include or query for policies defining data access rights and make a policy decision. In this case a file catalog service acts as a PDP.

3.2.2 Metadata Catalogs

The Metadata Catalog stores, as the name implies, metadata associated with actual data stored in the Grid. This allows users to search for data whose filename is not known to the user, e.g. for result data of a specific research experiment. Other metadata can be the date and time of a file, its size and all sorts of keywords describing the data. A Metadata Catalog allows for searching for files based on such metadata and returns a list of LFNs or GUIDs of those files whose metadata matches the query.

3.2.3 Replica Catalogs

In large Grid environments consisting of resources distributed across multiple sites it will be favorable to maintain multiple physical instances of one file at different locations at different sites transparently to the user. This means that the user does not know where replicas of a file in question are held. She only needs to address a file by its LFN or GUID and the replica catalog will locate the replica topologically closest to the destination of the file transfer to be initiated. The Replicy Catalog thus maps GUIDs to SURLs.

3.3 File Management Services

3.3.1 File Authorization Service

The File Authorization Service (FAS) holds the VO-level policies for file access in form of *Access Control Lists* (ACL). Each file registered by its GUID has an ACL in

form of a table associated with it in the FAS. An ACL contains policies for specific users or groups in regard to the file. The permissions defined in an ACL determine specific access rights such as whether a user is entitled, either by her ID or by her group membership, to read, write, create or delete a file.

Other services, in particular the service containing the PDP policing file access on the VO level, can query the FAS via the network for policies. The FAS can also be coupled with the PDP more tightly in form of a logical unit in the same program.

3.3.2 Storage Resource Managers

A Storage Resource Manager (SRM) is part of a Grid middleware which adds a layer of abstraction on Storage Elements between Grid data access protocols like GridFTP explained in chapter 3.4.1 and the low level parts of storage systems which can hold manifold types of data as described at the beginning of this chapter. It is also possible for Grid data access protocols to access storage directly but using a SRM in between has several advantages [SSG02]:

A SRM allows for dynamic allocation of storage to users in contrast to static quotas. As VOs become larger it might become unlikely that one user will have to copy data to a SE near a Computing Element on one particular site. Static disc quotas would thus be wasted storage space as users might never access disk space reserved for them. The use of group accounts instead of static mapping of Grid users to local users is another reason for using dynamic space reservation instead of static quotas as Grid users are not directly known to the local file system.

The following properties of SRMs are discussed in [SSG02], but in actual implementations some or all of these services can also be offered by other Grid services, e.g. by a File Transfer Service.

SRMs are also able to retrieve data needed for a specific job from other locations. If no replica of that file is stored permanently on the SRM it looks up the nearest replica and copies it onto the local file system. If another user requested that file before and a cached copy is still available on the local file system the SRM can use that file. This way SRMs can effectively reduce network traffic.

It is possible for SRMs to queue requests. If a resource is busy the client does not have to retry several times but the SRM queues the requests and processes them when the resource is available. This is especially useful if the SRM has a robotic tape server as storage which have by design longer access times than hard disks. The

same applies for temporary network failures in which case the SRM will also queue the request and retry for some time before sending a failure message to the user.

Finally, SRMs allow for streaming of large numbers of files. That means that a computing job might use a large number of data files, but uses only one of them at a time. The allocated space for a user might not be enough to store all of these files at once. The SRM first retrieves only the first few needed files and when the client application releases one file it will fetch the next one.

3.3.3 File Movement Services

File Movement Services add functionality which data transfer protocols as GridFTP described in chapter 3.4.1 do not offer. The two most common services are *multiple data object transfer services* and *reliable data transfer services*. The former allow for submission of a large batch of simultaneous file transfers and they initiate and monitor these file transfers. The latter monitor the states of queued data transfers and the progress of running transfers. Failed transfers will be restarted by the service for a specified amount of times. In case of permanent failure an error message is returned to the requester and the transfer is aborted.

File Transfer Service

A File Transfer Service (FTS) is responsible for adding an abstraction layer above pure data transfer protocols as GridFTP. A FTS keeps track of running transfers and tries to restart failed transfers. Specifically an FTS transfers data reliably from one Storage URL to another Storage URL. This behavior provides for a reliable data transfer service. Furthermore a FTS can schedule file transfers to optimize network usage. The services offered by a FTS imply that the service needs to be stateful, e.g. to restart failed transfers for a specific amount of times before aborting a job and sending an error message to the user. This is achieved by using a database backend in the FTS for storing state data. By this behavior the FTS offers a multiple data object transfer service.

File Placement Service

The File Placement Service is a service helping a job to decide where a file should be transferred to. If a job is scheduled to run on a site where no replica of the required

source data is available, the File Placement Service decides upon a suitable Storage Element on or at least near that site. It then delegates the execution of the transfer to the File Transfer Service and, if the FTS returns a success message, registers the new replica to the Replica Location Service.

3.4 Data Transfer Protocols

3.4.1 GridFTP

Grid environments need a fundamental data access and transport service. GridFTP allows for access to and transport between Grid nodes regardless of the actual implementation of the nodes and whether the data is transferred between two storage nodes or from or to a user.

GridFTP is an extension of the well-known File Transfer Protocol (FTP). For the use in Grid environments, FTP has been extended in several ways. Support for GSI Authentication has been added to allow Grid components to authenticate to GridFTP servers without the need for a username or a password, needed both for Single Sign On and delegation of rights. GridFTP connections do not have to be initiated by one of the nodes which transfer data, but can be initiated and monitored by third parties. This can be the user herself to transfer data from one node to another or it can be a specific file transfer service like RFT in the Globus Toolkit (see chapter 3.5.3). To allow for this feature additional monitoring capabilities have been implemented.

Furthermore, GridFTP supports partitioning data transfers to multiple striped TCP streams for increasing performance. In contrast to traditional FTP, GridFTP also supports encrypted connections between the participating nodes to meet security requirements imposed by many Grid communities.

3.5 Data Management in current Grid Middlewares

3.5.1 gLite with IO Server

Architecture

The central instance for data-management in gLite 1.5 is the IO Service, running the IO Daemon. The IO Service manages all stages of a file operation as can be seen in fig. 3.2. Data access using the IO Service exposes the files available within a VO in

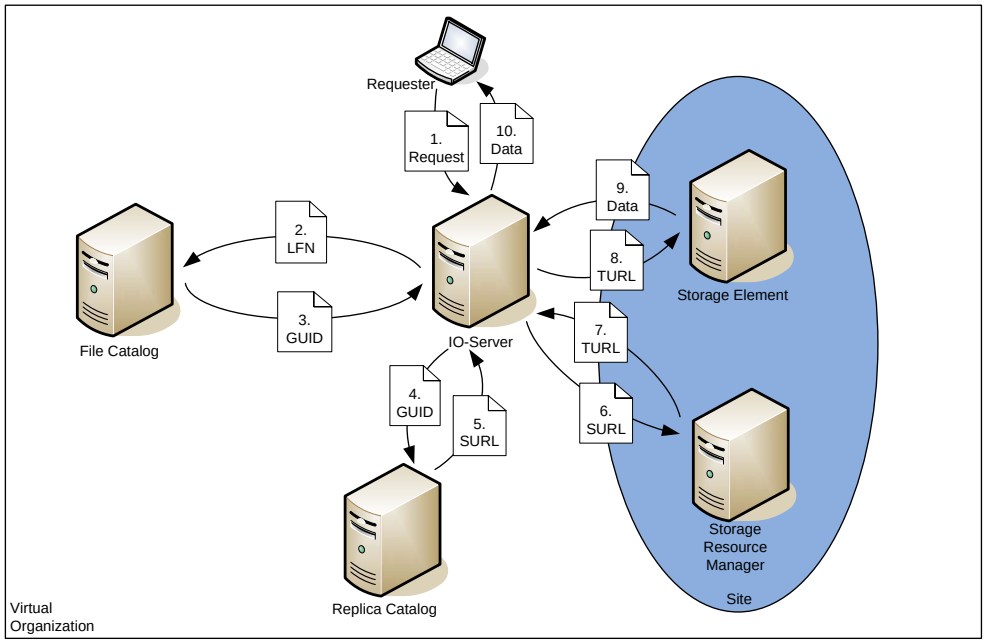

Figure 3.2: gLite Data Management

form of a Unix-like global file system. A file access request can be initiated directly by the user using the gLite command line tools which work similar to standard Unix commands such as ls, cd, mkdir and so on. These commands can also be part of a job and are thus initiated by another Grid component acting on behalf of the user presenting her credentials. The steps shown in figure 3.2 are in detail:

1. A request to access to a file is sent to the IO Service. The request contains the file in question (represented either by a LFN or a GUID), the access mode (e.g. read or write access) and the user credentials.

2. If a LFN is supplied it has to be resolved to it's GUID which is used internally to identify files. The mapping between LFNs and GUIDs is kept in the File Catalog. Furthermore the File Catalog includes the File Authorization Service used to decide wheter to allow or to deny the requested access. If a GUID is supplied by the user instead of a LFN only the authorization decision is made in this step.

3. The GUID is returned to the IO Service by the File Catalog if the access is granted, an error message otherwise.

4. Next the GUID needs to be translated into a Storage URL (SURL). A file can be replicated at multiple sites to allow efficient access to the nearest replica. The

SURL specifies a Storage Resource Manager (SRM) which has information about the physical storage location of the file.

5. The Replica Catalog returns a SURL to the IO Service.
6. The SRM is contacted by the IO Service to get a Transport URL (TURL) under which the file is accessible. The TURL contains the transport protocol (e.g. GridFTP), the name of the Storage Element (SE) and the path and filename of the requested file on the SE.
7. The TURL is returned to the IO Service by the SRM.
8. The TURL can now finally be used to access the file.
9. + 10. The file contents are relayed through the IO Service to the requester.

The File- and Replica Catalogs (and a possible Metadata Catalog not included in fig 3.2) can be combined within one service. This is the case with the FiReMan (*Fi*le and *Re*plica *Man*ager), one implementation combining the File Catalog and the Replica Catalog. The gLite model using the IO Server has a centralized policy repository included in the FiReMan service and the IO Service requests the policy information about a requested file access from it. It is possible to deploy more than one FiReMan per VO, but the model is still centralized as for each request only the FiReMan associated with the IO Service currently used will be queried. It is not possible to fetch policies from more than one FiReMan and combine the policies by any means.

Data Access Authorization

Authorization decisions are all made by the File Authorization Service, being queried by the IO Service, which thus acts as the PDP. In addition to the user's identity it is also possible to police resource access by the user's VO-affiliation or attributes asserted to the user by a VOMS server. All of these are available from the user's delegated credentials. As all file accesses have to be relayed through the IO Service no direct file access to the SRMs is available. The IO Service is the only entity entitled to access data on the SEs. This means that no authorization is done on the SRMs and SEs other than denying every access not originated by the IO Service and granting all accesses by the IO Service. All files on a SE thus belong to the IO Service and all information about access rights on user or group level is only available on the FiReMan server.

3.5.2 gLite with File Transfer Service

Architecture

The gLite Grid middleware version 3.0 does not officially contain the IO Service anymore. The IO Service is a major bottleneck in Grids based on earlier versions of gLite as all file transfers are not only initiated by it but the data itself is relayed through it. The new architecture introduced with gLite 3.0 is based on the *gLite File Catalog* and the *Grid File Access Library* (GFAL). GFAL is an API that hides direct access to the SRMs and catalogs to the client program. Client programs can be programmed in a way that allows programmers to implement their Grid tools similar to ones using only local file access. As File access authorization is done on the SRMs in this architecture, it is not possible to police file accesses based on the GUID. The GUID is not known to the SRMs and backward resolution from a SURL to a GUID is not possible on the SRMs without adding additional callbacks to the catalogs, which is not currently planned to be implemented in the near future. Fine-grained file access authorization as it is available by using the IO Service is thus not possible. For this reason some communities and organizations, including the RRZN, still rely on the IO Service instead of adopting the new architecture at this time.

Data Access Authorization

At present time the newly introduced architecture does not allow for VO-level policy decisions for file access. The only way to police file access is by using grid-mapfiles mapping Grid users to local users on the Storage Elements. This, compared to the IO Service, allows only for very coarse-grained policies. If Grid users are mapped to pool accounts instead of individual local users, policing file access on the VO-level becomes virtually impossible.

3.5.3 The GLOBUS Toolkit

Architecture

The Globus Toolkit does not use a central data-management component such as the IO Service in gLite. Data access can be performed directly by users via TURLs. If catalogs are available users have to use them directly to generate a TURL they can use for file access.

Furthermore, the Globus Toolkit includes a *Reliable File Transfer* (RFT) service which is a File Transfer Service as explained in section 3.3.3. This concept is comparable with the afore mentioned method used in gLite 3.0, leaving authorization decisions to the Storage Element.

Data Access Authorization

Data access is authorized in GT4 by the Storage Elements. The simplest approach is mapping of DNs to local users by a gridmap file and then use local UNIX file access control mechanisms decide on file access permission or denial.

A more sophisticated approach is the use of the Community Authorization Service (CAS) providing authorization information embedded within the user's credentials. This approach is based on a centralized policy repository using a push model to transport the policy to the storage element. The user requests an access token which is embedded into her credentials by the CAS server and only requests containing such a CAS extension inside the proxy certificate explicitly permitting the requested access may be authorized by the storage element.

Chapter 4

User-Based Restriction of Delegated Rights

It has been shown that user resource access rights can be exploited under certain circumstances in Grids using the Grid Security Infrastructure. This chapter is concerned with developing an approach for self-restriction of user rights to minimize potential damage imposed by an attacker. This concept is founded on User-Based Policies. Ways for their creation, encoding and propagation will be analyzed.

Furthermore, by using the foundation laid out in the preceeding two chapters, general design decisions about how and where to implement the PDPs and PEPs needed for enforcement of User-Based Policies are presented for different types of Grid implementations. In the next chapter an actual implementation of one of these designs will be presented.

4.1 Current Situation

As explained in the previous chapters Grid access authorization schemes police access to a requested resource by comparing the user's identity or other attributes, e.g. a VOMS extension, taken from delegated credentials and the desired resource access to policies. With the current implementation of the GSI, use of hijacked credentials will expose the full scope of access rights the user owns to the attacker, even those which are not relevant in the context of the current job because this particular job does not need to access all resources the user has the rights to access.

By using hijacked credentials an attacker is able to harm the user's interests in a multitude of ways:

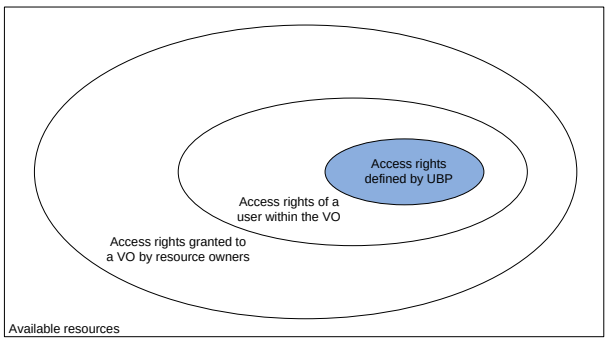

Figure 4.1: Hierarchy of Rights

- The attacker can utilize Grid resources in the name of the user. If the VO has a billing scheme for use of Grid resources the user will be billed for actions she did not perform herself.
- The attacker can spy out confidential data which the user has read access to. This is a serious threat to privacy for example in medical VOs with strict privacy constraints for stored data imposed by law and also allows for industrial or scientific espionage.
- The attacker can alter files the user has write access to. This can pose a serious threat as the attacker can sabotage projects relying on that data. If the manipulation remains unnoticed the project obtains wrong results derived from the altered data or, if the manipulation is noticed, the user whose credentials were abused might be suspected.
- The attacker can delete files the user has write access to. This also makes it possible for the attacker to sabotage projects or discredit the user whose credentials are used to do such actions.

This list of possible severe attacks shows that additional security mechanisms are required to ensure that security-sensitive communities will adopt Grid concepts for their computation needs. As shown in Chapter 2.5 it is not possible to completely eliminate the risk of such attacks without turning away from the GSI. The approach of this thesis is thus to find security mechanisms extending current implementations in a way that the threats listed above are minimized as much as possible.

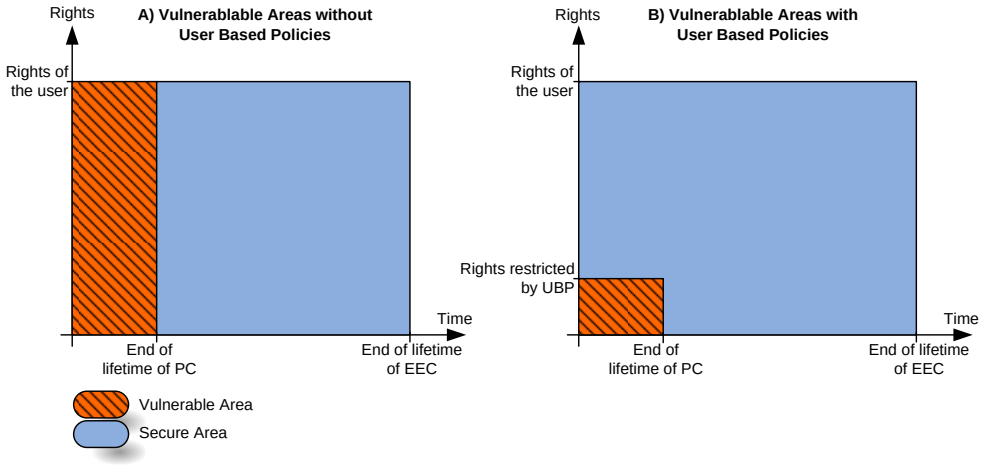

Figure 4.2: Vulnerable Areas in the Time-Rights-Space

4.2 User Self-Restriction of Rights

The solution discussed in this thesis to conquer this problem is to allow users to restrict the rights which they delegate to the Grid. In contrast to the global policies issued by the VO and site administrations these restricted rights do not determine the scope of rights a user has inside the Grid, but which of her rights the user wants to expose for current use. Figure 4.1 shows the scope of rights defined by such a User-Based Policy. The rights a user exposes by issuing a UBP are a subset of the rights the user is granted by the VO she is member of, which in turn are a subset of the rights a VO is granted by the sites contributing resources to that VO.

Figure 4.2 visualizes the advantages that self-restriction of rights by users employing UBPs brings forward. As can be seen in part A) without using UBPs the complete scope of rights of the user can be abused by an attacker until the end of the lifetime of the proxy certificate, resulting in an vulnerable area in Time-Rights-Space denoted by the orange hatched area. As the GSI effectively limits the vulnerable area in the time dimension through the use of short lived proxy certificates the blue colored area right of the vulnerable area can be considered as secure. In contrast to that there is no restriction in the rights dimension. The full scope of rights owned by the user can be accessed and thus also be abused within the time frame of the validity of the PC.

Part B) of fig. 4.2 shows the advantage of adding a rights restriction, comparable to the time restriction achieved by short lived PCs. The vulnerable area is now not only limited in the time dimension but also in the scope of rights dimension which means that there are less resources which can be compromised by an attacker.

As this thesis is concerned with data management in Grid computing the concepts described in the following sections are focused on UBPs limiting data access rights. Some, if not most, of these concepts can also be applied to restricting access to other Grid resources like computing nodes. This is albeit out of the scope of this thesis.

4.3 Properties of User-Based Policies

4.3.1 Contents of User-Based Policies

In order to allow a PDP to make an authorization decision based on an User-Based Policy it must first be defined what information the PDP needs to make this decision. First, it needs information about the actual access request to be decided upon:

- The target of a request, e.g. a LFN or GUID for file access or a Job ID (an unique ID issued for each job by the Grid middleware) for access to a computing resource.
- The requested mode of access, e.g. read, write or execute.
- The identity of the requester.

Second, the PDP needs the policy information containing the rules upon which to base the authorization decision. Each rule within a policy must include:

- The target to which the rule applies, e.g. a LFN, GUID or Job ID.
- If a target can be accessed in multiple ways, e.g. for writing and reading, for each of these access modes the policy must include whether the access is to be *granted* or *denied*.
- The identity of the requester whose access rights are expressed by the rule.

The information for the actual request can be derived from the context in which it occurs. The policy information, containing the three items enumerated above, has to be represented by the User-Based Policy. However, the identity of the user can be omitted inside the policy itself if the UBP is intrinsically tied to the credentials used to issue a job and thus implicitly stating the identity of the requester to whose requests the rules are to be applied to.

4.3.2 Encoding of UBPs

The User-Based Policies have to be encoded in an appropriate way to allow for their efficient and unambiguous creation and interpretation. They have to be created on the User Interface, transferred via the network and then be stored and evaluated by the respective PDPs.

If the PDP or the PEP are implemented as standalone services a network protocol is needed for communication between them and to the User Interface. If they are included in existing Grid services the UBP can also be piggybacked in some part of a job and thus must also be encoded in an appropriate way.

Current state-of-the-art technologies for exchanging structured data commonly rely on XML-based technologies. Policy information encoded in XML can be exchanged over a network by using web services which exchange data using HTTP or the XML data can be distributed in other ways in form of plain text data. The advantages of using XML are:

- Availability of specialized open non-proprietary XML-based policy markup languages, particularly XACML explained later in the next section.
- Tool support in most modern programming languages allowing for efficient parsing, validation and editing of XML data. Tool support is available for both generic XML data and specific policy markup languages.
- Extensibility for adding future features. XML allows by its design easy integration of new features by adding new elements.
- Simple data representation in plain text allows for easy inclusion of XML data in all kinds of containers without the need to define binary interfaces or additional encoding, e.g. using BASE64.

4.3.3 Creation of UBPs

As the name implies, UBPs are decreed and thereby created by the user herself. However, users of Grids cannot be assumed to be experienced in creating policies in some policy description language which by design are rather complex as can be seen in chapter 5.2.2 where the XACML Policy Language used for the implementation developed for this thesis will be described. It is thus beneficial to hide this complexity from the user by presenting an, if possible, graphical user interface. Even

for experienced users this will increase productivity as creation of policies by hand, e.g. with a text editor, is time-consuming and error-prone. A command line-based interface without all features explained below can however still be provided on User Interfaces to the Grid, e.g. if no graphical desktop is available.

In respect to UBPs concerning file access it is possible to present the user a graphical file browser following well known primitives for browsing local file systems. Since LFNs are simulating a hierarchical file system with directories and files, the user can be presented a tree-view of that virtual file system and in one pane and the files contained inside the currently selected directory in another pane. This way of displaying files is well known e.g. from the Microsoft Windows Explorer.

In such a graphical representation only files accessible by the user should be displayed. This way an user would not be able to include policies granting access to files which she is not allowed to access by global policies. Since such an included policy would not be effective anyway, as the access would still be denied by the VO-level PDP, this is not a security requirement but it would prevent confusing the user.

As explained above besides the target of the policy, i.e. the filename, all possible access modes have to be granted or denied by the user. Again the user interface should hide impossible options from the user. This can be done for example by using checkboxes which, when checked, grant an access mode. If a user does not have write access to a file this checkbox should be greyed out so that the user cannot grant write access within her UBP but still add a policy granting read access.

If the user has more than one Grid identity, i.e. she has more than one EEC with different DNs installed on the User Interface, she will have to choose one DN to be included as the user identification inside the UBP. If, as will be explained in the next chapter, the user identity is implicitly stated by using a proxy certificate containing the DN of the user as a container for the UBP, this information can be completely omitted.

If a user creates her User-Based Policy by hand arbitrary rules can be added to it. However, it must not be possible for the user to grant access to a resource which she does not have any VO-level access rights to. This can be achieved by policing access request by the conjunction of the results of the evaluation of both the global access rights and the rights granted by the UBP. That means that both policy evaluations have to *grant* an access request and if at least one *denies* access the requested access has to be denied. This way a user might add rules to her UBP allowing access to

resources she does not have the rights to access, but they will not grant any additional access rights and are thus ineffective.

4.4 Evaluation and Enforcement of UBPs

4.4.1 Push, Pull and Agent-Based Implementation

In a *push*-based implementation the services processing resource access do not have to explicitly ask for the policy information by querying some other entity but the policy information is pushed to them by the requester, normally included in the request itself. The advantage of this approach is that the PEP does not have to access the PDP itself. In respect to designing a model for evaluation and enforcement of UBPs the push model is favorable because the short-lived and tightly coupled User-Based Policy information has to be propagated to the Grid anyway before or while submitting a request.

Uploading the UBP to a repository for access from PDPs in a *pull*-based implementation would only add an additional service and several additional steps for uploading and getting the UBP from or to that service without any reasonable benefit. It is also not possible to let the User Interface itself offer the UBP for other services as, following the Single Sign On and Delegation of Rights paradigms, the UI might be turned off during execution of a job.

The third possible implementation model, the *agent* model, is not useful if a new agent would have to be designed and implemented from scratch. As the requested file access has to be policed by the VO-level authorization service *after* the authorization decision based on the UBP is made, the agent service would have to authenticate to the VO-level service using the user's credentials anyhow. There would thus be no benefit over a push-based implementation where the PDP issues a token permitting access.

If, on the other hand, the VO-level authorization is realized in an agent-based model like gLite's IO Server[1] we have a different picture: The PDP and PEP policing resource access by UBPs can then be integrated directly into that service and if both, the UBP and the VO-level policies, grant access, the existing agent then can realize the actual access to the data as it already does without the UBP concept.

[1] in fact it is a mixture of an agent-based and a pull-based architecture: from the macro perspective the IO Service is an agent deciding upon and enforcing file transfer requests, but from a closer perspective the IO Server does not make the authorization decisions itself but delegates them to the File Authorization Service, which is a feature of a pull-based architecture.

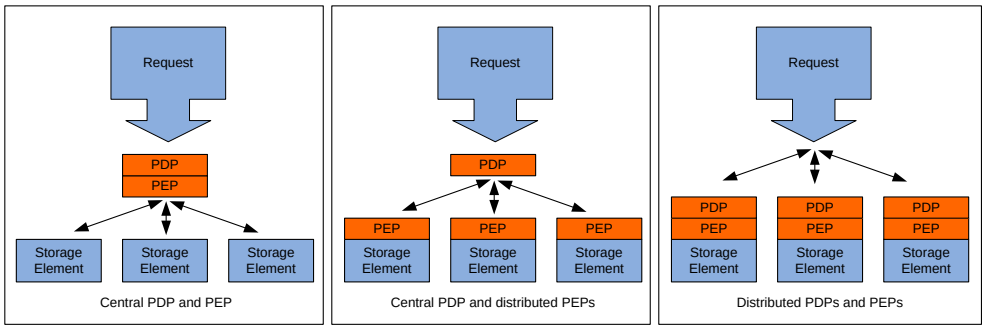

Figure 4.3: Possible Deployments of the PDPs and PEPs

4.4.2 Deployment of the PDP

The PDP and the PEP can both be deployed in several ways as shown in fig. 4.3. User-Based Policies impose the need for an additional PDP and the corresponding PEP as these added policies have to be evaluated and enforced. In current Grid implementations based on gLite or the Globus Toolkit at least two stages of PDPs are already present: First, the PDP on the VO Level deciding whether the user has access to e.g. one specific file and secondly the PDP on the site level deciding whether the resource is shared with the VO by the site administration. The second PDP is on a very coarse level allowing access to all members of a VO. The PDP on VO level is more fine-grained by making decisions based on the access rights of the user whereas the new PDP to be added is even finer-grained by deciding on the policies issued by the user herself.

It is thus reasonable to implement the new PDP in a way that the UBP is evaluated and enforced before the other policies. This way the finest-grained policy is evaluated first and the coarsest-grained one last. If access is already denied by the UBP the request does not have to be transmitted to and evaluated by the other PDPs at all. This chained arrangement of PDPs is represented in figure 4.4.

Another fact to be considered is the availability of the right representation of the file: If the UBP is based on LFNs and GUIDs but resource access is realized by SURLs and TURLs, callback mechanisms to the catalogs have to be implemented to do a reverse-resolution to find out the LFN or GUID to a TURL or SURL used to access a file.

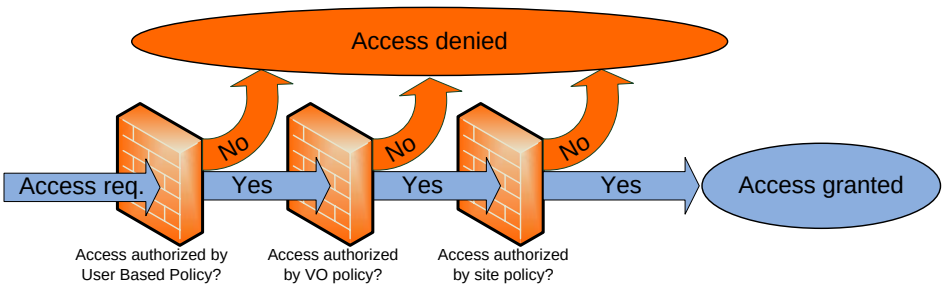

Figure 4.4: Chained Policy Decisions

Centralized versus Distributed Deployment

The PDP can principally be deployed as one central service on the VO level or there can be multiple distributed PDPs, e.g. one on each Storage Element. The only pre-condition that alway has to be met in all implementations is the availability of all information necessary for making the policy decision as described in section 4.3.1.

The ideal deployment model of the PDP for evaluating UBPs depends on the location of the PDP for VO-level policies as the UBP shall be evaluated before the VO-level policies. If that service is implemented in a centralized way like gLite's IO Server or Globus' CAS, the PDP for UBPs should also be deployed the same way, either on the same node or topologically close to that node. Deployment in that way minimizes the risk of adding an additional performance bottleneck to the Grid. It would also be impractical to evaluate the UBP at the resource level if the VO policies are evaluated by a centralized service and, at the same time, evaluate the UBP before the VO-Level policies.

As UBPs typically include considerably less rules compared to policy data for a whole VO stored in a File Authorization Service, evaluation of the UBP will not significantly increase the complexity of a service already deciding VO level policies. The PDP can thus be implemented inside the service providing the VO-level PDP. Implementing a dedicated service for evaluating the UBPs on a dedicated server only increases network traffic and latencies with no justifying benefit.

If VO-level policies are evaluated on the Storage Elements both implementations are possible: The SEs could initiate a callback to a centralized service on VO level for evaluating a UBP or the PDP could be implemented on the SE itself. Again, due to the required availability of the necessary information and the limited complexity that

is added by evaluating UBPs compared to VO-level authorization the PDP for UBPs is best implemented inside or near that service, i.e. on or near the SEs.

The PDP presented in chapter 5 for gLite-based Grids using the gLite IO Server for managing data access has been added directly to the IO Daemon and is thus a centralized approach as described above.

4.4.3 Deployment of the PEP

In a push- or pull-based infrastructure the Policy Enforcement Point must be implemented on the Storage Element if no callback mechanism is implemented to fetch the UBP from. Otherwise it would be possible to circumvent the PDP by accessing the data on the SEs directly. It will either evaluate whether the presented token authorizes access to the requested file (push model) or it will ask the appropriate PDP whether the access is to be granted or denied. It must not be possible by any means to access the data on the SE without passing through the PEP, i.e. by bypassing it in some way.

In an agent-based implementation on the other hand the PEP should be located on the agent realizing data access. As the data on the SEs is accessed with the rights of the agent service and not using the delegated user's rights the PDP cannot be located on the Storage Elements.

4.5 Propagation of User-Based Policies

The UBP has to be propagated to the PDPs involved in permitting or denying resource access. The UBP is by its nature tightly coupled with one specific job (though many jobs of similar nature might share the same requirements regarding resource access and thus the same UBP might be reused). A UBP is for the same reason only expedient for a limited time frame.

It is consequently the obvious choice to include the UBP into the credentials used to issue a job compared to making it available by other means, e.g. by a repository service. In order to prohibit the exchange of the UBP by an attacker by a new one granting additional rights cryptographic measures have to be taken, e.g. by signing the UBP with the user's EEC private key. This way an attacker would not be able to create a new UBP granting additional rights to be used in conjunction with the hijacked user's credentials.

4.5.1 Implementing User-Based Policies in GSI

As discussed above, pushing User-Based Policies from the user to the corresponding PDPs together with a job, i.e. in the credentials used to issue the job, is preferable to implementing additional stand alone services and protocols. In Grid environments using a X.509-based authentication and authorization scheme like the GSI the UBP can be embedded in the proxy certificate the user creates for delegation of her rights to the Grid. X.509 certificate extensions as described in 2.2.2 are a concept for inclusion of arbitrary data into certificates.

This way of propagating the UBP has several advantages over other possible implementations:

- The extension containing the UBP will be signed together with the rest of the proxy certificate by the private key of the user's issuing EEC. This means that if an attacker hijacks the user's credentials it is not possible for the attacker to exchange or alter the UBP in any way without invalidating the whole credentials.
- The user's credentials are propagated throughout the Grid to every service that needs to access resources on behalf of the user. This means that no additional protocols and services have to be implemented to propagate the UBP.
- As the UBP is automatically pushed to existing PDPs it is an easy task to add the new PDP evaluating UBPs to the existing system or daemon. This way additional callouts to external systems for acquiring the UBP can be avoided.

Technical details about the inclusion of XACML-encoded UBPs into X.509 proxy certificates are explained in depth in section 5.2.3.

4.6 Properties of UBPs for File Access Authorization

4.6.1 Specific Requirements for File Access

File access in general and especially in Grid environments have some special requirements regarding access authorization: Files can be accessed in different modes whose number and types depend on the implementation, e.g. read and write, of the components used for data access. Furthermore one file can be referenced in multi-

ple ways, e.g. by LFN or GUID. All reasonable combinations of file references and access modes must be representable by the chosen data format for the UBP.

As discussed in chapter 3.2 a file can be referenced by a symbolic link, its LFN or GUID and by Storage- and Transfer URLs. Which of these possible file references are usable directly by the user depends on the implementation and architecture of the VO. For instance users must not be able to use SURL and TURLs for data access if the gLite IO Service is used for managing data access. As the access authorization decision is made by the IO Service, users accessing files directly by SURLs or TURLs would be able to bypass VO-level file access authorization. GUIDs can be exposed to the user as files might be deleted from the File Catalog but still be present on a Storage Element. In this case the user can still access that file by its GUID.

The following access modes are used in this thesis, similar to those used in [FePe05]:

- *Read* access allows an user to read the contents of a file. It is possible to grant users only access to a subset of the file contents, e.g. by stating accessible bytes. This is however uncommon for remote access to files.
- *Write* access allows an user to alter the contents of a file. Like read access, write access might be granted to only a part of a file.
- If write access does not include creation of new files, *write-once* privileges can be issued for allowing creation of a file.
- If deletion of files is not implied by write access, *delete* privileges can be issued for allowing deletion of a file.

Other environments, especially ones that do not rely on files as the elementary data item but e.g. on relational databases, might have different or additional access modes. These are out of the scope of this thesis and are thus not considered.

The use of an extensible data format as XACML for storing and exchanging policy information allows for easy adaption to different Grid environments. While retaining the fundamental structure of the policy document new variants of file descriptors and different file access modes can easily be added. Additionally compatibility to existing components evaluating the UBP can be achieved if the XACML parser is implemented in an appropriate way. Unknown access modes or file descriptors, for instance, could be ignored by such a PDP if that cannot allow for unauthorized file access.

4.6.2 Limitations of the concept

The usefulness of the concept of user self-restriction of their rights for file access depends on the actual environment it is implemented in. It is most effective if users generally have rights for accessing more files than typically needed to be accessed in single jobs. One example for such an environment is the medical community: Medical doctors generally have access to many files, each containing medical data about exactly one patient. If the doctor issues a job to a Grid where these data are stored she will most likely need to access only the file of the patient currently under treatment. In such a case the doctor could issue a UBP where only access to that specific file is granted. All other files would not be accessible using her current credentials.

In other environments on the other hand, where users only have access to a very limited number of files or typical jobs require access to all or most files the user has access rights to, the usefulness of issuing UBPs for restricting file access would greatly be reduced. Further work is needed to offer such communities additional benefit from using UBPs. Some possible steps for further development of the concept are presented in the outlook section at the end of this thesis.

Chapter 5

Implementing UBPs in the gLite Grid Middleware

In the previous chapter different concept for implementing user-based policies for resource access in general and for data access in particular have been presented. As part of this thesis a working prototype has been implemented to demonstrate the use of UBPs for self-restriction of user rights. In this chapter the design decisions made while developing the architecture and used techniques for programming of the prototype are presented and discussed.

To ensure a minimal but feature complete implementation of the user self restriction of rights using user-based policies a thorough requirements analysis had to be performed. The aim of the project is to extend the existing gLite Grid infrastructure at the RRZN in a way that users are able to effectively self-restrict their rights delegated to the Grid. As explained in section 3.5.2, the RRZN still uses the IO Service instead if the new architecture introduced with gLite 3.0. As especially security sensitive communities are dependent on fine-grained access control the implementation developed here is also based on an infrastructure using the IO Service.

5.1 Requirements

The software implemented for this is concerned with policing data access only, albeit further development for additional resources is planned for further master theses. This implies an extensible approach allowing for easy adaption to future extensions. The software to be developed shall be a fully functional prototype. This means that all functional requirements have to be met. Non-functional requirements, in particu-

lar security requirements, do not have to be met to the full extent. The prototype is meant to prove the concept of user-based policies for user self-restriction of rights. It is not meant for production use.

5.1.1 Functional and Non-Functional Requirements

The following *functional requirements* for the creation of proxy certificates including user-based policies have to be met:

- The software to be developed must be compatible to the installed gLite Grid middleware version 3.0.0 at the RRZN.
- Users must be able to express their self-restriction of rights to be delegated to the Grid in a standardized and extensible way if form of a User-Based Policy.
- The file access policies must allow for targets specified by either their LFN or their GUID.
- It must be possible to address multiple files in a single policy by issuing LFNs including wildcard characters. This does not apply to files referenced by their GUID because GUIDs are hash values with no explicit meaning.
- Each file access policy must express either *grant* or *deny* for the four access modes *read*, *write*, *write-once* and *delete* as explained in section 4.6.1.
- The UBP must be propagated to the Grid and it must be made available on all resources concerned with policing data access. For future extensibility it should also be available on other components concerned with policing resource access.

The *non-functional* requirements for the creation of proxy certificates including user-based policies are in detail:

- The inclusion of a UBP into a proxy certificate should be handled in a user-friendly way.
- Extension of existing gLite user tools shall be considered first before implementing new tools from scratch.

The PDP and PEP policing the user-based policies have to meet the following *functional requirements*:

- The existing Grid infrastructure has to be extended in a way that UBPs are evaluated and enforced.
- By no means it shall be possible to perform a file access that is not explicitly granted by the UBP, if present.
- Credentials not containing a User-Based Policy must be accepted with a default policy granting all file accesses. This means that if a user decides not to delegate only a sub-set of her rights the full scope of her rights must be usable. If however a UBP is present, the default policy for files not policed by the UBP must be *deny*.
- The aforementioned behavior must be configurable allowing for UBPs to become obligatory. This means that if no obligatory UBP is available all accesses must be denied.

Furthermore the *non-functional requirements* for the PDP and PEP services are:

- Evaluation and enforcement of UBPs should not increase the execution time for policing a data access by more than 100%, i.e. it should not take longer than policing the VO-level access rights of the user.
- Additional network overhead by the PDP and PEP should be avoided.
- In case of a denied access a meaningful error message should be returned to the user.
- The software should be programmed in a way that minimizes possible bugs that allow bypassing the PEP.

5.1.2 Grid Middleware

The prototype has been developed for a gLite 3.0-based environment relying on the IO Server for data management. The existing gLite testbed installed at the RRZN has been extended by two new nodes to install the developed software without interfering with the existing systems: Firstly, a new User Interface has been set up to install and test the extended user tools for creating and including UBPs in generated proxy certificates. Secondly an additional IO Server has been set up to allow for installing and testing the extended IO Service.

5.1.3 Programming environment

The software has been developed on a computer with standard Intel x86 compatible PC hardware as also all hosts part of the gLite testbed are based on Intel x86 compatible hardware.

Scientific Linux 3.04 (SL3) has been chosen as the operating system. SL3 is 100% binariy compatible with Red Hat Enterprise Linux 3 and is officially supported by gLite 3.0. As the gLite testbed and the two added computers are also running this operating system, it was chosen for the development system to ensure compatibility and easy deployment of the developed software from the develeopment system to the actual gLite servers.

The software presented in this chapter has been developed in C++ using the Eclipse IDE version 3.2.0. For additional support of C++ the Eclipse C/C++ Development Toolkit (CDT) version 3.1.0 has been used.

The ant-based build system of gLite has been used to build the packages. The gLite build system results in RPM packages ready for installation using the Red Had Package Manager (RPM) used by Scientific Linux.

5.2 Architecture

5.2.1 The Big Picture

The system level overview sequence diagram in figure 5.1 shows a typical file access in gLite enhanced by the components which were developed as part of this thesis. The blue components are part of the standard gLite-based environment. The orange parts have been added from scratch and the hatched blue-orange parts are enhanced standard gLite components. The design decisions that led to this architecture and implementation were made on the basis of the discussions in the previous chapter:

- The User-Based Policy will be distributed by including it into the user's proxy certificate. This way the UBP will be available at all stages of a Grid job where authorization decisions based on the user's identity and rights are made.
- The PDP will be integrated on the IO Service. See 4.4.2 for further details. The user's credentials including the UBP and the file to be accessed and the access mode are available on the IO Server and this way no additional network

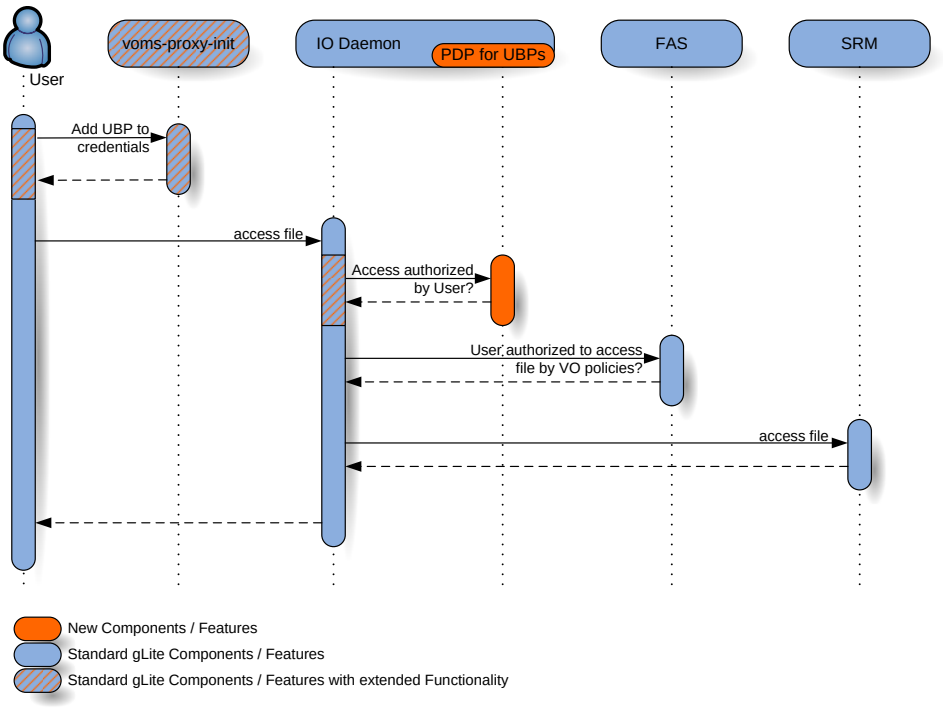

Figure 5.1: Sequence Diagram of a gLite File Access

traffic is added to the infrastructure. Also existing error handling mechanisms available for dealing with denied file accesses can be reused to handle accesses denied by the UBP.

- The PEP will also be integrated into the gLite IO Service. The gLite IO Service is realizing data access by an agent-based architecture. As discussed in chapter 4.4.3 in such an environment the PEP is best implemented on that agent service, thus in this case on the IO Server.

The user that can be seen in the diagram can be the actual user herself or a service acting on behalf of the user. If a service acts on behalf of the user by using her delegated credentials the first step of creating a proxy certificate is not applicable. The rest of the diagram remains the same in both cases.

5.2.2 The XACML gLitePolicy Data Model

It has already been discussed why the XACML policy markup language is ideal for expressing user-based policies. This section describes the exact semantics which are understood by the IO Daemon. The format is chosen in such a way that future addi-

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <PolicySet xmlns="urn:oasis:names:tc:xacml:2.0:policy:schema:os"
3              xmlns:xacml="urn:oasis:names:tc:xacml:2.0:policy:schema:os"
4              xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
5              xsi:schemaLocation="urn:oasis:names:tc:xacml:2.0:policy:schema:os
6                                  http://docs.oasis-open.org/xacml/2.0/XACML-CORE/
7                                  schema_files/access_control-xacml-2.0-policy-schema-os.xsd"
8              PolicyCombiningAlgId="identifier:rule-combining-algorithm:first-applicable"
9              PolicySetId="gLiteUserPolicy">
10    <Target />
11    <Policy ...>
12     ...
13    <Policy ...>
14  </PolicySet>
```

Figure 5.2: Basic Structure of a XACML Document

tions of policies not directed at file access but at restricting access to other resources, e.g. computing nodes, is possible without breaking compatibility with the IO Service.

The XACML syntax is described by a XML schema available from [OAS06]. This document describes in a standardized way what elements are mandatory for a valid XACML document. As XACML offers features not needed for this implementation not all features XACML offers are used here.

Figure 5.2 shows the basic structure of an XACML document for expressing UBPs. In line 1 the document is specified as a XML document with UTF-8 character encoding. An UBP expressed in XACML starts with a *PolicySet*. XACML allows for different root elements for XACML documents. One using *PolicySet* is capable of including an arbitrary number of policies. In a UBP policies for multiple files and other resources must be represented, thus a *PolicySet* is used to wrap these policies.

The lines 2 to 7 define used namespaces and the location of the XML schema document. This is useful to identify documents and to access the XML schema describing the document syntax. With the XML schema it is possible to automatically validate XML files for correctness. If a document is correct according to a XML schema it is called *valid*. A file with a generally correct XML syntax is called *well-formed*. In lines 2 and 3 the same namespace is declared two times, as the nameless default namespace and as a namespace with the qualified name *xacml*. This is neccessary to allow for XPATH[1] evaluation of the contents of the policy in the PDP. Another solution would be to only use qualified namespaces instead of the declared default namespace.

The *RuleCombiningAlgId* attribute in line 8 defines an algorithm for combining rules concerning the same target. One rule might grant access, another rule deny it. The three algorithms explained in 2.4.1 are available.

[1] http://www.w3.org/TR/xpath

```
 1     <Policy PolicyId="FilePolicy1"
 2             RuleCombiningAlgId="identifier:rule-combining-algorithm:deny-overrides">
 3      <Target>
 4       <Resources>
 5        <Resource>
 6         <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
 7          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
 8            lfn:///path/to/my/file
 9          </AttributeValue>
10          <ResourceAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"
11                                       DataType="http://www.w3.org/2001/XMLSchema#string" />
12         </ResourceMatch>
13        </Resource>
14       </Resources>
15      </Target>
16     <Rule Effect="Permit" RuleId="Read">
17      <Target>
18       <Actions>
19        <Action>
20         <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
21          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
22            Read
23          </AttributeValue>
24          <ActionAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"
25                                     DataType="http://www.w3.org/2001/XMLSchema#string" />
26         </ActionMatch>
27        </Action>
28       </Actions>
29      </Target>
30     </Rule>
31     <Rule Effect="Deny" RuleId="Write">
32      ...
33     </Rule>
34     <Rule Effect="Deny" RuleId="Write-Once">
35      ...
36     </Rule>
37     <Rule Effect="Deny" RuleId="Delete">
38      ...
39     </Rule>
40    </Policy>
```

Figure 5.3: Example of one Policy in a XACML Document

In line 10 an empty *Target* element is specified. This element is mandatory as its presence is demanded by the XML schema. This element is supposed to identify the entity the rules in the XACML document apply to, i.e. the user whose access rights the rules describe. As, in this implementation, the identity of the user is already specified by the DN of the proxy certificate the XACML document is embedded into, there is no need to make use of that element. It is thus only kept as a self-closing element without any further attributes to ensure the policy remains a valid XACML document.

The following lines are an arbitrary number of elemental policies policing access to exactly one file or, if wildcards are used, to all files matching the regular expression. Single *Policy* elements are discussed in the next paragraphs. Finally in the last line the *PolicySet* Element is closed and the XACML document ends.

The policy items themselves can be of arbitrary types. The PDP on the IO Server will only evaluate those having a *PolicyID* attribute (figure 5.3 line 1) of the form *FilePolicyXX* where XX is a consecutive numbering of the included data policies starting from "1" denoting the first policy concerning data access.

One *FilePolicy* as shown in figure 5.3 consists of a *Target* starting in line 3 and ending in line 15. The target describes the file the policy is targeted at either in form of a LFN

or a GUID encoded as a string (line 7) which should be formatted in a way the PDP understands, otherwise the rule will have no effect. XACML allows for the inclusion of more than one target in a policy but that feature is not used by this implementation. For each file a new policy has to be added. The use of wildcards allows for multiple files to be referenced by a single policy. The sequence of *Resources* thus contains exactly one *Resource* (lines 5-13).

In line 6 is defined that the target stated in the policy is a match to a target of a request and the rule is thus applicable if they are referenced by the same string (*funcion:string-equal*). This is necessary to match an actual request to a policy rule.

Associated with a target are a number of rules describing which actions to allow or deny on that target. In file policies four modes of access are defined by rules: *Read*, *Write*, *Write-Once*, and *Delete* as defined in 4.6.1. Each of these access modes can either be *Denied* or *Permitted*. If one or more of these rules are missing that rule should be interpreted by the PDP as denied.

In figure 5.3 only the rule for reading the targeted file is completely depicted (lines 16-30), the other three are abbreviated. Read access is allowed by setting the *Effect* attribute of the *Rule* element to *"Permit"*. Write, Write-Once and Delete access to the file are denied by setting the attribute to *"Deny"*.

The XACML data format allows for different correct syntaxes expressing the semantically same policy. For the PDP in the IO Server it is mandatory that the above explained format to express UBPs is used. Other formats, even if syntactically correct XACML and semantically equivalent, will not be understood by the PDP.

Furthermore, the policies expressed using this XACML format have the following properties. These properties are founded on the actual implementation of the PDP in the IO Service and are thus not necessarily portable to other resources potentially policed by UBPs.

- Either a LFN or a GUID may be used to describe a file in a UBP. The descriptor in the policy must be preceded by either "lfn://" or by "guid://".
- It is possible to use wildcards in LFNs used in the policies: A "*" stands for an arbitrary string and a "$" for exactly one character.
- "Deny" rules override "Permit" rules. This can be used to e.g. permit read access to all files in a directory using the "*" wildcard but exclude some files again. This could be overridden by stating a different rule combining algorithm

in the XACML policy, but the current implementation does not evaluate this and always uses "Deny-Overrides".

- Always all four possible access modes ("Read", "Write", "Write-Once" and "Delete") should be stated for each file entry inside your policy. However missing access modes are defaulted to "Deny" by the PDP.
- Each one of these access modes must be set to "Permit" or "Deny" where "Permit" means a requested access has to be granted and "Deny" means it has to be denied if the file referenced by the policy is to be accesses using that specific access mode.
- In the certificate chain inside the credentials may only be one (or no) certificate containing a UPB extension. If two or more are found, access will be denied. Otherwise an attacker could derive another proxy certificate using the certificate and private key containing a forged UPB and thus extend the rights granted by the legitimate user. This might be extended in the future to allow multiple UBPs in subsequent proxy certificates, but in such a case it must not be possible to broaden the scope of rights specified by the first UBP by later ones. Such additional UBPs may only be used to further limit the scope of delegated rights. This could be used by Grid services to delegate only the needed rights for the current action to other services and not the full scope of rights the user delegated for the whole job.

5.2.3 Propagation of UBPs using Certificate Extensions

As already discussed user-based policies are to be distributed with the user's credentials delegated to the Grid. This is possible by using X.509 certificate extensions explained in section 2.2.2 to embed the UBP into the user's proxy certificate which is part of the delegated credentials.

To unambiguously identify a policy extension it must be tagged with a unique ID, the Object Identifier (OID) as explained in section 2.2.2. The RRZN already has an assigned base OID which is 1.3.6.1.4.1.18141. To identify an extension containing user-based policies ".3.100.1.1" has been added. 3 is the branch of the tree assigned to the *Lehrgebiet Rechnernetze*, 100 for Grid use, 1 for expressing User-Based-Policies and the final 1 denotes that this is the first iteration of UBPs. If the XACML data format will be changed at some time in the future this last number can be in-

cremented. This concludes to the complete OID to "1.3.6.1.4.1.18141.3.100.1.1" for identifying user-based policies as implemented in this thesis.

To enable such an implementation the user program for creating proxy certificates has to be extended in a way that it accepts a XACML document as a parameter and includes that document as a X.509 certificate extension to the PC that it creates. This has been done by adding the functionality to the voms-proxy-init tool part of the gLite VOMS tools.

5.2.4 Evaluation and Enforcement of UBPs

The standard IO Server implements an agent-like architecture for authorizing file access using a File Authorization Service containing file access policies issued by the VO. As discussed in the previous chapter in such a configuration it is useful to tightly integrate the PDP for the UPBs with the existing PDP within the IO Daemon. The added PDP is located inside the daemon right before the existing PDP. In case of denial of access the error methods already existing in the daemon are reused. They are implemented in a way that it is possible to return a precise error message to the user stating at exactly what stage the access has been denied.

Reasons for denial caused by UBPs are:

- The access can be denied if the above mentioned rule combining algorithm comes to that outcome based on the rights stated in the UBP.
- The access can be denied if no UBP is present and it is configured on the PDP to be mandatory.
- The access can be denied because no rule for the targeted file exists and the default rule is to deny access.
- The format of the UBP can be incorrect. If the policy is not expressed by a well-formed XML document, the document is not valid XACML or it does not follow the semantics described in chapter 5.2.2, in general if the XACML parser can not parse the presented data, all accesses will be denied.

5.3 Implementation

5.3.1 The gLite Build System

The gLite Middleware source code currently (September 2006) consists of more than 1.3 million lines of code and is written in at least 17 different programming languages [GLI06]. This code is distributed among more than 200 modules. Software as complex and heterogeneous as the gLite middleware needs a sophisticated build system to help developers compiling the source code to binary and to take care of dependencies among gLite components and to external libraries.

The gLite build system is based on ant[2], a Java-based build system developed by the Apache Foundation. Ant is used as a meta-build system as the modules themselves are, if written in C or C++, mostly based on the gnu make system.

To compile a gLite component from source the following steps are necessary. In this example the data module is checked out from CVS and the IO Server being part of the data module is built. Other modules are checked out and built accordingly.

```
1 > export CVSROOT=":pserver:anonymous@jra1mw.cvs.cern.ch:/cvs/jra1mw"
2 > cvs co -r GLITE_RELEASE_3_0_0 org.glite
3 > cvs co -r GLITE_RELEASE_3_0_0 org.glite.data
4 > ant -f org.glite.data/project/glite.data.csf.xml
5 > cd org.glite.data
6 > ant io-daemon
```

In the first line the environment variable $CVSROOT is set to the path of the anonymous CVS server of the gLite project. The second command checks out the base packages necessary for every gLite module whereas the third command is responsible for retrieving the desired gLite module. The ant command in line 4 does not yet compile the sources but it checks the build system for presence of necessary components such as a Java runtime environment. Furthermore it downloads a repository from the gLite web servers containing all necessary shared libraries and tools needed for compiling the desired gLite component. After this step finished successfully a component of the module can be build by changing to the org.glite.data directory (or the respective directory for other modules) and start the build process with another ant command as in line 6 of the above example.

[2] http://ant.apache.org/

The ant-based build process builds RPM packages ready for use on the desired target platform. These packages can be installed by executing

```
> rpm -Uvh packagename.rpm
```

on the target system.

General directions for how to add new components to the gLite build system are out of the scope of this thesis. The next sections will deal with the gLite build system only as far as necessary to build the subsystems with the enhanced functionality. Further information is available in [DiM05].

5.3.2 Extending voms-proxy-init

The voms-proxy-init command is part of the gLite VOMS package and is invoked by Grid users to create a proxy certificate. In the version used for this thesis a *policy* argument is already implemented which stores arbitrary information from a text file inside an extension of the newly created proxy certificate. As this policy is reserved for future use by the gLite developers a new *userpolicy* argument has been implemented which technically does the same thing as the already implemented argument, but it tags the extension with a different OID assigned in the namespace of the RRZN as explained in section 5.2.3. OIDs are defined in the sources in the file *org.glite.security.voms/src/ac/init.c*. The specific mechanisms for creating the extended proxy certificates can be seen from the source code included on the accompanying CD-ROM in the file *org.glite.security.voms/src/client/vomsclient.cc*.

As all external libraries needed for creating certificates including extensions are already used by the existing program, the build system did not have to be changed to successfully compile the extended VOMS tools including the voms-proxy-init tool.

5.3.3 Implementation of a PDP and PEP in the IO-Daemon

New Classes, Methods and Variables

The IO Daemon which is the main part of the IO Service has been extended to accept, evaluate and enforce user-based policies in XACML format embedded in a X.509 certificate extension of the proxy certificates presented by entities requesting file access.

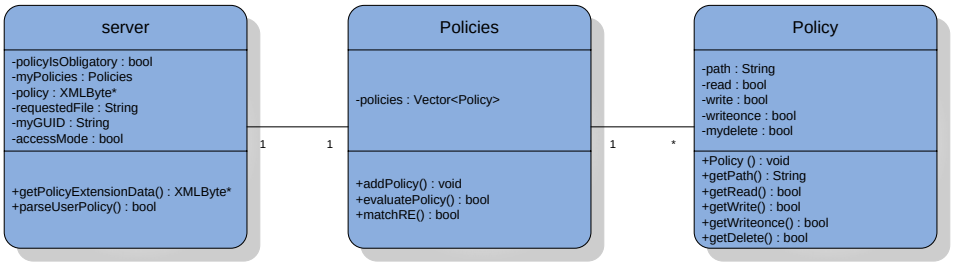

Figure 5.4: Class Diagram of the enhanced IO-Daemon

The class diagram in figure 5.4 shows the two new classes as well as variables and methods added to the existing IO Daemon. Here only the most important variables are explained. Additional helper or temporary variables can be seen and are explained in the source code itself.

The following variables have been added to the IO Daemon server class available on the accompanying CD-ROM in the file *org.glite.data.io-daemon/src/server.cxx*:

- *policyIsObligatory*: This boolean specifies the behavior of the PDP in case no UBP is issued by the user. If set to false, all accesses are granted by the PDP, if set to true, all accesses will be denied and an error message will be returned to the requester. This option is not configurable but must be changed to the desired value before compilation.
- *myPolicies*: This variable in the IO Daemon holds an instance of the new Policies class.
- *policy*: This variable holds the XACML policy data in form of an array of XMLBytes returned by the method *getPolicyExtensionData()*.
- *requestedFile*: This variable holds the name of the file which access to is requested.
- *accessMode*: This variable holds an integer between 1 and 4 where 1 stands for *read*, 2 for *write*, 3 for *write-once* and 4 for *delete* access.

New methods added to the IO Daemon server class are:

- *getPolicyExtensionData()*: This method reads the user's credentials from the its temporary location on the IO Server's hard disk and scans the embedded certificates for extensions with the appropriate OID and returns the contents of a found extension in form of a XMLByte array.

- *parseUserPolicy()*: This method parses the XMLByte array returned by the *getPolicyExtensionData()* method and adds the found policies to the *myPolicies* object.

The new class *Policies* is a container for the atomic policies and has methods for adding and evaluating policies. The class can be found on the CD-ROM in the files *org.glite.data.io-daemon/src/Policy.h* and *policy.cpp*. The variables and methods are in detail:

- *policies*: This variable is a *Vector* of instances of the *Policy* class. For every policy found in the UBP a new policy is added by the *parseUserPolicy()* method in the IO Daemon.
- *addPolicy()*: This method adds a new policy to the policies vector. The LFN or GUID and the policy for the four access modes must be passed to this method as parameters.
- *evaluatePolicy()*: This method evaluates if a given request is granted by the rules in the policy vector. Both, the LFN (if available) and the GUID of the requested file have to be passed to this method together with the desired mode of access. The method returns true in case the access is granted and false if access is denied.
- *matchRE()*: This method is used by the *evaluatePolicy()* method for matching the LFN of the request to the LFNs stored in the policies. As a policy LFN may contain wildcards this methods returns true if both are identical or the requested LFN matches the expression from the policy containing wildcards and it returns false otherwise.

The new class *Policy* contains information about exactly one policy rule. For every *FilePolicy* element parsed from the XACML user-based policy document a new instance of this class is created and added to the *policies* vector in the *Policies* class. The class can be found on the CD-ROM in the files *org.glite.data.io-daemon/src/Policies.h* and *Policies.cpp* The variables and methods are in detail:

- *path*: This string holds the path of the file the policy is targeted at. This can be a LFN with or without wildcards or a GUID. LFNs have do be prefixed by "lfn://" and GUIDs by "guid://".

- *read, write, writeonce and mydelete*: These boolean variables are set to true if the corresponding access mode is granted for the file stated by the path variable and to false if access is denied. The *mydelete* variable is called this way as delete is a reserved keyword in C++.
- *Policy()*: This is the constructor method of the Policy class. It initializes a newly created policy object with the pathname and access modes passed to it by the *addPolicy()* method from the *Policies* class.
- *getPath()*: This get method returns the LFN or the GUID respectively.
- *getRead(), getWrite(), getWriteonce() and getDelete()*: These get methods return true if the access mode is granted and false if not.

Adaption of the gLite Build System

Two additional external libraries had to be used by the *parseUserPolicy()* method of the extended IO Daemon to parse the XACML data. First, xerces-c[3] has been used to parse the serialized XACML data from the certificate into a DOM tree, an internal tree-based data structure representing the XACML data. Second, xalan-c[4] has been used to extract the relevant information using XPATH expressions from the DOM tree. The exact procedure can be seen in the source code.

As additional external libraries are used by the extended IO Daemon and two new classes were added the build system had to be extended to compile the new classes and include the external libraries. First, the Makefile found in *org.glite.data.io-daemon/src/Makefile.am* had to be changed to compile the two new classes. Second, two entries in the file *org.glite.data.io-daemon/configure.ac* were added to make the compiler aware of the two new external libraries and third, the file *org.glite.data.io-daemon/project/runtime.dependencies* had to be altered so that the RPM packages generated by the build scripts include dependencies for the two added external libraries.

Collaboration of the Components

The sequence diagram in figure 5.5 shows a typical request for read access to a file. Other access modes only differ in the called get-method in step 6. A request is handled by the following steps:

[3] http://xml.apache.org/xerces-c/
[4] http://xml.apache.org/xalan-c/

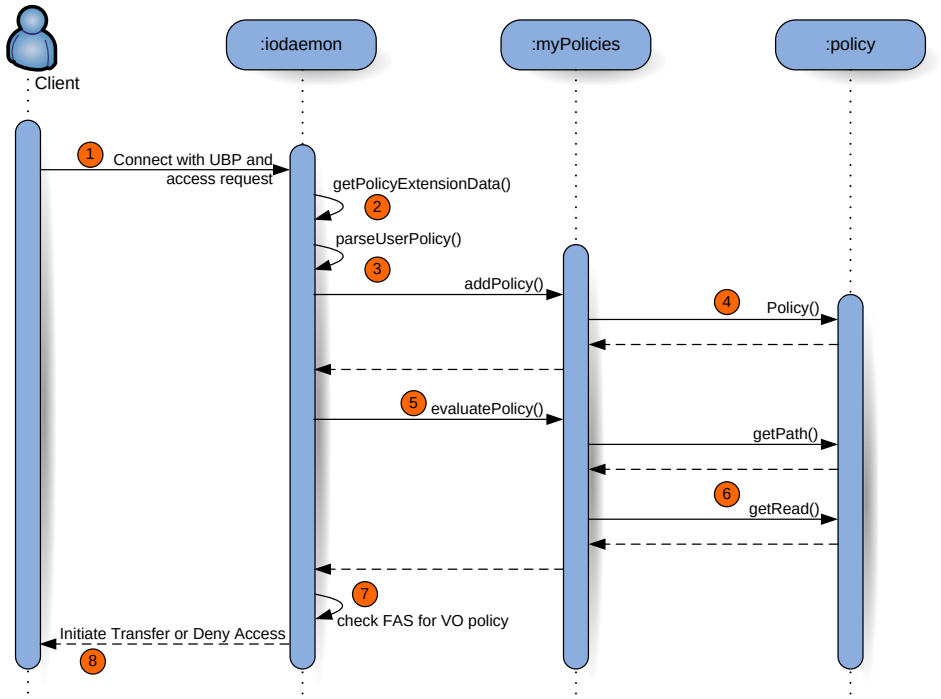

Figure 5.5: Sequence Diagram of the enhanced IO-Daemon for a read access request to a file.

1. An access request from a requester (the user herself or a service acting on behalf of the user) arrives at the IO Daemon. The request includes the user's credentials, the file to be accessed and the access mode.

2. The XACML content of the certificate extension including the UBP is extracted from the credentials.

3. The extracted XACML data is parsed and a *myPolicies* object is created which includes all the atomic policies.

4. A newfound file policy is added by creating a new policy object initialized with the file descriptor (LFN or GUID) and the four access modes are set to "Deny" or "Grant". This step repeats until all file policies from the XACML document are extracted.

5. When all policies are extracted the requested access is evaluated by calling the *evaluatePolicy()* method with the file and access mode requested by the client as parameters.

6. All available policies are iterated through and if the requested filename and the filename from the policy match the desired access mode, i.e. grant or deny, is evaluated using the "deny-overrides" algorithm.

7. If the access is granted by the UBP the File Authorization Service is queried for VO level policies for the requested file. If the UBP already denies access this query is omitted and the client revieves an error message.

8. If the FAS grants access, the data transfer is initialized or, if not, an error message is returned to the client.

Limitations of the enhanced IO Service

As the implementation is a working prototype and not a final product intended for production use, it has several limitations. Some of these are based on the time constraint of six months for this thesis, others are due to specific peculiarities of the gLite IO Service and will not apply to other implementations of user-based policies for restricting data access in Grids.

The limitations are in detail:

- The XACML file will not be checked against the XML-Schema defining a valid XACML document. This means every well formed XML document with entries matching the XPATH statements in the Policy Decision Point of the IO-Daemon will be considered as valid policies.

- The different Data Policies must have consecutive numbering in their PolicyIDs, i.e. FilePolicy1, FilePolicy2 and so on. If one is omitted the following ones will not be evaluated.

- As the gLite IO-Daemon at the moment does not allow to write to existing files, "Write" is not supported. Setting this property to "Permit" or "Deny" will not have any effect. If "Write-Once" is set to "Permit" creation of the file will be granted.

- The code has not been audited for security. It is a proof-of-concept and it will probably have bugs allowing buffer-overflows and similar security threats. It should not be used it in production environments.

```
 1   > voms-proxy-init -userpolicy policy.xml
 2   Your identity: /O=GermanGrid/OU=UniHannover/CN=Ralf Groeper
 3   Enter GRID pass phrase:
 4   Creating proxy ............................................................ Done
 5   Your proxy is valid until Sun Oct  8 01:43:35 2006
 6   > glite-put test /tmp/testfile -s gliteio://ioadmin2.gridlab.uni-hannover.de:6666
 7   [glite-put] Total 0.00 MB       |====================| 100.00 % [0.0 Mb/s]
 8
 9   Transfer Completed:
10
11     LFN                    : /tmp/testfile
12     GUID                   : 00871248-92f0-1527-ac5d-824b0728beef
13     SURL                   : srm://dcache1.gridlab.uni-hannover.de:8443/[...]
14     Data Written [bytes]   : 5
15     Eff.Transfer Rate[Mb/s] : 0.000001
16
17   > glite-rm /tmp/testfile  -s gliteio://ioadmin2.gridlab.uni-hannover.de:6666
18   ERROR  Cannot unlink remote file. Reason is: 'AIO Error: access denied on remote host'
19   ERROR  Cannot unlink file. IN THE SERVER, reason is: 'Access Denied by user policy!'
20   Cannot Unlink Remote File /tmp/testfile. Error is "AIO Error: access denied on remote host (code: -24)"
```

Figure 5.6: Example session with user-based policies

5.4 Example of a File Access Policed by UBPs

Accessing a file with a User-Based Policy granting the access does not differ from normal file access in the standard gLite middleware. Only if an access is not granted by the UBP or another error, e.g. an error due to a non-valid XACML format of the UBP, occurs, the new functionality becomes visible to the user. In the example session in figure 5.6 a UBP allowing for creation (write-once) of the file */tmp/testfile* but denying its deletion is used. In line one a new proxy certificate is created including a UBP from the file *policy.xml*. In line 6 a file called test in the current directory of the User Interface is transferred to a newly created file within the Grid with the Logical Filename */tmp/testfile*. As can be seen in the lines 7 to 15 the transfer is successful. In line 17 the file is tried to be deleted again. This action does not succeed as the UBP does not allow the deletion of the file. This can be seen in line 19 by the error message *"Access Denied by user policy!"*.

5.5 Additional Considerations

5.5.1 Applying UBPs to gLite-based Grids without the IO Service

As discussed in section 5.1.2 the gLite IO Service is not part of the standard gLite distribution anymore as of version 3.0. While some communities still rely on the IO Service for policing file access, in the long term this functionality will be transferred from the central service the IO Server is to a distributed model where VO-level file access authorization will be done on the Storage Elements. This means that user-based policies also need to be evaluated and enforced somewhere else.

As explained above, on the IO Server all relevant information for making an authorization decision on self-restricted rights are available without the need for external callbacks. This also applies to file accesses directly on the SEs, as the user's credentials, information about the file to be accessed and the desired mode of access are avaliable there just like they are on the IO Server. Opposed to the IO Service, the request will not be initiated by stating the LFN or GUID of a file but by a SURL or TURL. As UBPs are to be enforced based on LFN and GUIDs, it will become necessary to implement a callback from the SRMs to the catalogs to resolve the SURL of the requested file back to its GUID or LFN to make a policy decision. Here further research is needed as outlined in the outlook section at the end of this thesis.

5.5.2 Interaction with MyProxy

The MyProxy credential store is used in Grids to store user's certificates and private keys. This has two benefits: First, users do not have to keep their private keys secret themselves. As many users might not be security aware or even computer savvy, key hygiene is a major factor to make Grids using the GSI secure. Second, a MyProxy server allows for dynamic proxy renewal. If jobs need longer than the lifetime of the used proxy certificate that job will fail, e.g. when results are to be transferred from the Worker Node to a Storage Element after computation finishes.

If the user issued a proxy certificate containing a User-Based Policy, after a proxy renewal this policy will not be part of the credentials any more because technically the PC's lifetime is not extended but a completely new PC is generated by the MyProxy server.

With some additional programming efforts this problem can be solved: A PC created by the MyProxy Server does not necessarily have to be derived from the user's EEC but it can be derived from another proxy certificate. The client tools the user uses to create and embed the UBP must thus create a proxy certificate containing the UBP and upload this PC to the MyProxy Server. This PC must have a lifetime longer than a standard proxy certificate used for credentials delegated to the Grid. If the PC uploaded to the MyProxy Server has a lifetime of e.g. two times the default lifetime of a normal PC, the Grid can get a new proxy certificate from the MyProxy service three times. The UBP will always be included in the newly created credentials as the chain of trust always has to be kept complete and thus the PC containing the UBP will be returned to the Grid together with the newly created PC.

Chapter 6

Conclusion and Outlook

6.1 Summary

This thesis started with a general outline about Grid security and Grid data management laying out the foundation for the following chapters. The Grid Security Infrastructure has been explained and data management concepts discussed. Furthermore, data management in two major Grid middlewares, namely gLite and the Globus Toolkit, has been compared.

In chapter 4 possible attacks based on hijacked credentials have been evaluated and a solution for minimizing potential damage done by such hijacked credentials has been presented. This solution involved the concept of user-based policies which allow users to restrict the rights they delegate to the Grid in form of their credentials, thus effectively narrowing the value hijacked credentials have for an attacker.

Finally a real life implementation of user-based policies for the gLite Grid middleware has been presented. A XACML-based data format for expressing UBPs has been developed and the gLite tool for creating proxy certificates has been extended in a way that allows for the inclusion of such UBPs. Furthermore the IO Service, which is the central service managing gLite data management, has been extended to evaluate and enforce such UBPs. The working prototype implemented as part of this thesis is available, both as source code and in binary form for Scientific Linux 3, on the accompanying CD-ROM for further review and testing.

6.2 Conclusion

In this thesis a review of the Grid Security Infrastructure has been presented, a security weakness identified and a solution for minimizing the damage caused by exploiting that weakness has been discussed and implemented. The implemented solution has proved to be a viable approach for allowing users to control the rights they delegate to Grids.

The concept of policing data access by user self-restriction of their rights has shown to be a first step in the right direction. If the following steps outlined in the outlook prove to be equally successful, self-restriction of user access rights delegated to Grids in the form of user-based policies embedded in proxy certificates will hopefully help to make the Grids of tomorrow more secure and thus more attractive to potential users.

6.3 Outlook

The implementation presented in this thesis is a working prototype for demonstrating the use of user-based policies. If the Grid community approves the concept, the software from this thesis can serve as a starting point but a new implementation is necessary by applying a complete software engineering concept including in particular security reviews.

In chapter 2, RFC 3820 has been introduced as an effort to standardize proxy certificates. RFC 3820 compliant PCs allow for the inclusion of policies in a standardized way. If the gLite middleware should adopt that standard at some time, it will be useful to utilize that way to embed user-based policies. Also, if the concepts presented in this thesis are to be implemented in the Globus Toolkit, which already supports RFC 3820 compliant PCs, making use of these standardized features is advisable.

As the IO Service is to be deprecated by the gLite community its replacements have to be extended with PDPs deciding upon and PEPs enforcing user-based policies. This will, as planned today, most likely concern the SRMs on the Storage Elements or the gLite File Transfer Service.

Also the concept of users restricting the rights they delegate to Grids by issuing user-based policies has to be extended to policing more resources than only files. Especially restricting the rights for access to computing resources is desirable.

Another dimension of restriction can be added if not only actions on resources are granted or denied by UBPs but also the set of entities allowed to perform these actions can be restricted. For instance a job could be registered with its requirements to a Grid meta scheduler as gLite's WMS. The WMS reserves a free computing resource fitting to the job's requirements. The WMS can now tell the client e.g. the DN of that server's EEC. The user can now limit her granted file accesses to exactly that computing element and thus effectively prevent an attacker to even access the files access to is granted by the UBP if the attacker does not have access to that computing resource.

In addition to gLite other Grid middlewares can be enhanced by introducing user-based policies. First of all the Globus Toolkit can be targeted as it also relies on the GSI and thus it should be an easy task to transfer the technique from gLite to Globus. Other Grid middlewares not using the GSI can be evaluated both, for new ideas how to enhance security in GSI-based middlewares and how to enhance them by implementing UBPs or other concepts from GSI.

Furthermore it is desirable, if more and more Grid resources are to be policed by UBPs, to develop a shared library with a predefined API in contrast to the monolithic approach taken in this thesis. Such a library would greatly simplify the extension of further components, both for data management and for other resources.

Appendix A

User's and Administrator's Guide

A.1 VOMS Tools

A.1.1 Installing the VOMS Tools

Installing the Binary Packages

The provided precompiled RPMs are for Scientific Linux 3.0 on i386 architecture.

To install using the RPM simply copy it to a preferably already set up and working gLite UI node and install it as root using

```
> rpm -Uvh glite-security-voms-1.6.16-8.i386.rpm [--force]
```

Note: –force may be needed if you get any errors about the new packages conflicting with the already installed ones.

Installation from Source

To install from source you first need a complete checkout of the glite data management package from CVS. Attention: you need more than 700MB free disk space and a decent internet connection to download all of this!

The following steps are required to compile the user-based policy aware IO-Daemon. They work on my system, but you might have to adapt some steps to your configuration.

1. Create a new workspace directory to put everything into. This directory will be called $WS from now on.

2. set the CVS-environment variable:

   ```
   > export CVSROOT=":pserver:anonymous@jra1mw.cvs.cern.ch:/cvs/jra1mw"
   ```

3. Check out the base gLite sources for version 3.0:

   ```
   > cvs co -r GLITE_RELEASE_3_0_0 org.glite
   ```

4. Check out the gLite security sources for version 3.0:

   ```
   > cvs co -r GLITE_RELEASE_3_0_0 org.glite.security
   ```

5. Get all the dependencies and additional sources for building the gLite security module:

   ```
   > ant -f org.glite.security/project/glite.security.csf.xml
   ```

   This will take some time.

6. Replace and add the files from the user-based policy package:

   ```
   > tar -xzfv glite-UBP-VOMS-tools.tar.gz
   ```

   (Some files are added, others replaced. The paths of the files from the archive have to be preserved. See below.)

7. Change to the org.glite.security directory:

   ```
   > cd org.glite.security
   ```

8. Start the build process for the IO-Daemon with dependencies:

   ```
   > ant voms
   ```

   This will take some time. You will need the packages "docbook-utils" and "docbook-utils-pdf" to successfully compile this, so you might have to install them, e.g. by using apt-get.

9. Copy the RPMS from $WS/dist/sl30/i386/RPMS to your UI and install them using the instructions above.

Modified Files

The following files part of the standard gLite sources have been edited and are part of this archive:

```
$WS/org.glite.security.voms/src/client/vomsclient.h
$WS/org.glite.security.voms/src/client/vomslient.cc
$WS/org.glite.security.voms/src/ac/init.c
```

A.1.2 Using the VOMS Tools

To include a UBP use the -userpolicy option of grid-proxy-init:

```
> grid-proxy-init [other options] -userpolicy=example.xml
```

The policies from example.xml are added as a non-critical extension to your proxy certificate.

You can submit jobs or issue data management commands and the UBP will be supplied to the Grid components automatically inside your proxy certificate when submitting a job.

A.2 IO Server

A.2.1 Building and Installing the IO Server

Installing the Binary Packages

The provided precompiled RPMs are for Scientific Linux 3.0 on i386 architecture.

To install using the RPMs simply copy them to a preferably already set up and working gLite IO-Server and install them as root with

```
$ rpm -Uvh *.rpm [--force]
```

Note: –force may be needed if you get any errors about the new packages conflicting with the already installed ones.

Now restart gLite using:

```
$ /etc/init.d/gLite restart
```

Installation from Source

To install from source you first need a complete checkout of the glite data management package from CVS. Attention: you need more than 700MB free disk space and a decent internet connection to download all of this!

The following steps are required to compile the User-Based Policy aware IO-Daemon. They work on my system, but you might have to adapt some steps to your configuration:

1. Create a new workspace directory to put everything into. This directory will be called $WS from now on.
2. Set the CVS-environment variable:

   ```
   > export CVSROOT=":pserver:anonymous@jra1mw.cvs.cern.ch:/cvs/jra1mw"
   ```

3. Check out the base gLite sources for version 3.0:

   ```
   > cvs co -r GLITE_RELEASE_3_0_0 org.glite
   ```

4. Check out the gLite data management sources for version 3.0:

   ```
   > cvs co -r GLITE_RELEASE_3_0_0 org.glite.data
   ```

5. Get all the dependencies and additional sources for building the gLite datamanagement:

   ```
   > ant -f org.glite.data/project/glite.data.csf.xml
   ```

   (This will take some time.)

6. Replace and add the files from the User-Based Policy package:

   ```
   > tar -xzfv glite-UBP-DataManagement.tar.gz
   ```

   Some files are added, others replaced. The paths of the files from the archive have to be preserved. See below.

7. Change to the org.glite.data directory:

   ```
   > cd org.glite.data
   ```

8. Start the build process for the IO-Daemon with dependencies:

```
> ant io-daemon
```

This will take some time.I f you get error messages about missing $WS/repository/axis/1.1 simply create a symlink from axis 1.2.1, this seems to work fine:

```
> ln -s $WS/repository/axis/1.2.1 $WS/repository/axis/1.1
```

9. Copy the RPMS from $WS/dist/sl30/i386/RPMS to your IO-Server and install them using the instructions from above.

Added and Modified Files

The following files have been edited or added to the standard IO-Daemon sources and are part of this archive:

```
$WS/org.glite.data.io-daemon/src/server.cxx                [changed]
$WS/org.glite.data.io-daemon/src/Policy.h                  [new]
$WS/org.glite.data.io-daemon/src/Policy.cpp                [new]
$WS/org.glite.data.io-daemon/src/Policies.h                [new]
$WS/org.glite.data.io-daemon/src/Policies.cpp              [new]
$WS/org.glite.data.io-daemon/src/Makefile.am               [changed]
$WS/org.glite.data.io-daemon/configure.ac                  [changed]
$WS/org.glite.data.io-daemon/project/runtime.dependencies [changed]
```

A.2.2 Using the IO Server

The enhanced IO-Daemon does not need any further configuration compared to the standard gLite IO-Daemon. Simply follow the instructions from the INSTALL file included in this archive.

The following error messages can be returned to a requester by the IO Server in case of a denial of access:

- *"Access Denied by user policy!"*: This means that the requested access is not granted by the User-Based Policy included in the credentials delegated with the request.
- *"Access Denied by user policy! Multiple proxy certificates contain policy information!"*: This error message occurs if more than one proxy certificate from

the certificate chain inside the delegated credentials contain a user-based policy extension. In this case access is denied as an attacker might have created an additional proxy certificate containing an User-Based Policy extension to obtain additional access rights the legitimate user did not want to expose.

- *"User Policy is not a valid XML document"*, *"Problem parsing user policy! Malformed XML?"* and *""Problem parsing user policy! Exception caught."*: These error messages mean that the supplied User-Based Policy is not a well-formed XML document. It can thus not be parsed and all accesses are denied.
- Furthermore, if the IO Service is configured to regard UBPs as obligatory, *"Access Denied by user policy! User Policy is obligatory but not available!"* means that no UBP has been supplied.

Appendix B

Contents of the CD-ROM

The CD-ROM contains the following items:

- This thesis in PDF format.
- Selected items from the bibliography (if available in electronic form).
- The binaries for the voms tools and the IO Daemon (both for obligatory and non-obligatory UBPs).
- The source files of all files changed or added to the standard gLite sources.
- User's and Administrator's Guide (identical to Appendix A).

Bibliography

[DiM05] A. DI MEGLIO. *EGEE Developer's Guide For The gLite EGEE Middleware*. https://edms.cern.ch/document/468700 (2005).

[FePe05] A. FEICHTINGER, A. PETERS. AUTHORIZATION OF DATA ACCESS IN DISTRIBUTED STORAGE SYSTEMS. In Grid Computing (2005). The 6th IEEE/ACM International Workshop on Grid Computing, pages 172-178.

[FoKe04] I. FOSTER, C. KESSELMAN. *The Grid: Blueprint for a New Computing Infrastructure, Second Edition*. Morgan Kaufman Publishers (2004). ISBN: 1-558-60933-4

[FVCG00+] S. FARRELL, J. VOLLBRECHT, P. CALHOUN, L. GOMMANS, G. GROSS, B. DE BRUIJN, C. DE LAAT, M. HOLDREGE, D. SPENCE. *AAA Authorization Requirements*. Request for Comments 2906, Network Working Group (2000). http://www.ietf.org/rfc/rfc2096.txt

[GLI06] EGEE ENABLING GRIDS FOR E-SCIENCE IN EUROPE. *EGEE > gLite > Quality Assurance: SLOC*. http://glite.web.cern.ch/glite/project/sloc.aspx (29.09.2006)

[HPFS02] R. HOUSLEY, W. POLK, W. FORD, D. SOLO. *Internet X.509 Public Key Infrastructure: Certificate and Certificate Revocation List (CRL) Profile*. Request for Comments 2459, Network Working Group (1999). http://www.ietf.org/rfc/rfc2459.txt

[ITU02] ITU-T. *Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation*. ITU-T Recommendation X.680 (2002). http://www.itu.int/rec/T-REC-X.680-200207-I/en

[ITU05] ITU-T. *Information technology - Open Systems Interconnection - The Directory: Public-key and attribute certificate frameworks*. ITU-

T Recommendation X.509 (2005). http://www.itu.int/rec/T-REC-X.509-200508-I/en

[MAMG99+] M. MYERS, R. ANKNEY, A. MALPANI, S. GALPERIN, C. ADAMS. *X.509 Internet Public Key Infrastructure: Online Certificate Status Protocol - OCSP*. Request for Comments 2560, Network Working Group (1999). http://www.ietf.org/rfc/rfc2560.txt

[MyPr06] BOARD OF TRUSTEES OF THE UNIVERSITY OF ILLINOIS. *MyProxy Credential Management Service*. Board of Trustees of the University of Illinois (09/05/06). http://grid.ncsa.uiuc.edu/myproxy/

[OAS06] ORGANIZATION FOR THE ADVANCEMENT OF STRUCTURED INFORMATION STANDARDS (OASIS). *Language (XACML) Version 2.0 Policy Schema*. http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-policy-schema-os.xsd

[PWFK02+] L. PEARLMAN, V. WELCH, I. FOSTER, C. KESSELMAN, S. TUECKE. *A Community Authorization Service for Group Collaboration*. http://www.globus.org/alliance/publications/papers/CAS_2002_Revised.pdf

[SSG02] A. SHOSHAMI, A. SIM, J. GU. *Storage Resource Managers: Middleware Components for Grid Stroage*. 19th IEEE Symposium on Mass Storage Systems, San Diego, CA (2002).

[TWEP04+] S.TUECKE, V. WELCH, D. ENGERT, L. PEARLMAN, M. THOMPSON. *Internet X.509 Public Key Infrastructure (PKI): Proxy Certificate Profile*. Request for Comments 3820, Network Working Group (2004). http://www.ietf.org/rfc/rfc3820.txt

[VCFG00+] J. VOLLBRECHT, P. CALHOUN, S. FARRELL, L. GOMMANS, G. GROSS, B. DE BRUIJN, C. DE LAAT, M. HOLDREGE, D. SPENCE. *AAA Authorization Framework* Request for Comments 2904, Network Working Group (2000). http://www.ietf.org/rfc/rfc2904.txt

[Wel05] V. WELCH. *Globus Toolkit Version 4 Grid Security Infrastructure: A Standards Perspective*. http://www.globus.org/toolkit/docs/4.0/security/GT4-GSI-Overview.pdf (2005).