

Authorization of Data Access in Distributed Systems

Derek Feichtinger

*CERN, CH 1211 Geneva 23, Switzerland and
Paul Scheerer Institut Zuerich, Switzerland*

Andreas J. Peters

CERN, CH 1211 Geneva 23, Switzerland

I. MOTIVATION

A common idea in GRID systems is to organize and group users in virtual organizations. Users get virtual roles assigned which define their permissions with respect to the datamanagement system as a member of the virtual organization.

A difficulty arises with a one-to-one mapping between roles in virtual organizations and local user accounts defined on the operating system level. The user community is highly dynamic and a synchronization of accounts on thousands of distributed computers is not achievable and even not desirable.

For the longterm future one can expect, that the concept of virtual organizations and virtual users will get supported at the kernel level of operating systems. At present this is not the case.

Therefore the proper handling of authorization for file creation, read, write and deletion operations is not trivial. Rules have to be defined on a file level and must be kept outside the data storage itself (usually in a file catalogue) following the previous considerations.

To produce a virtual role in a virtual organization we have to identify the user and apply the appropriate authorization rules before any execution of a data access request.

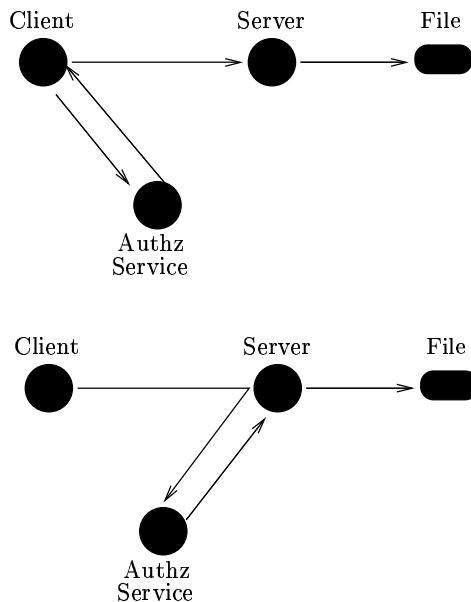
A. Client- and Server-Side Authorization Callbacks

Data access in general is described by a simple client server model. For security reasons it is not possible to handle the authorization on the client side like it is done in some present implementations (GFAL access library or the rfiio protocol). A secure framework needs to perform authorization at the server-side. Therefore data server need to communicate with an authorization service which defines for every file the access permissions and the physical location of a file.

The key terms in a GRID context are:

- the file catalogue, which defines the access rules
- the logical file name, which is used by a client to indicate a file to perform an access with

FIG. 1: Client- and Serverside Authorization



- the transport url, which is used by the data server to identify a logical file name from the catalogue with a physically stored file on the storage system.

B. The Use of Authentication as Authorization

In every datamanagement system client authentication has to be done. So it is obvious to think to merge authorization into the authentication part.

Grid-Proxy Certificates or VOMS-Certificates are nowadays used as the standard way for client authentication. The client certificate subject can be mapped with an external service or map file to a virtual role in a virtual organization. A VOMS Proxy Certificate can allow to put the possible virtual roles connected with a certain certificate subject directly into the certificate so that no external mapping is needed anymore. In a datamanagement system one could define virtual

groups or acls which are published in a VOMS proxy certificate if they are also used as file system groups on the storage system.

For file authorization the VOMS server has to add to every proxy certificate all groups or acls which define a permission for that specified user. The data server can afterwards compare them with the local used group permissions and authorize access based on this group permissions.

This works well, if the number of access control rules is low. If a virtual organization needs fine grained access control it not feasible.

This picture leads to an important disadvantage: a VOMS service gets connected with the file catalogue since they have both to know about defined access rules and the storage system has to provide a one-to-one mapping for virtual groups and physical groups.

C. Decoupling of Components in GRID Environments

Experience has shown that the maximum decoupling of components is a robust basis for a working GRID implementation and leaves system modularity and flexibility to exchange single components.

In a modular system with GRID-to-GRID interfaces it is a highly desirable goal to have a completely GRID service and configuration independent datamanagement and storage system because it allows to be used by all GRID implementations and by local users.

D. Sharing of Dataservers between VOs

In our case of VO datamanagement even the previous described server-side authorization callback framework is imperfect. F.e. the four big LHC experiments propose four different solutions of a file catalogue (or an authorization instance). So it is impossible to use the same datamanagement server unless the server can switch on the fly to different authorization servers depending on the VO the client is assigned to. Unfortunately the currently existing implementation of server-side authorization in glite-IO is not able to perform this context switching. Moreover the performance of such an external service callback has to be considered critical because the data server thread or process will be blocked for the time the service request is executed.

E. Requirement Definition for GRID Data Access Authorization

Following the above considerations we formulate an authorization mechanism with the following charac-

teristics:

- Usage of (proxy-) certificates as client authentication
- Usage of GRID specific file catalogues to define access permissions on the file level
- Hide storage information from the client
- Independence of data servers from grid services
- Possibility to share data servers between VOs
- High performance framework without authorization deadtime for the data server

These requirements have brought us finally to a token-like mechanism with access envelopes for authorization of data access.

II. THE PRINCIPLE OF FILE ACCESS ENVELOPES FOR AUTHORIZATION

For each file access a client requests an access envelope (token) for a single or a group of files from the responsible catalogue. This access envelope is send with every file access to the data server. The server authorizes the access based on the information specified in the access envelope.

This schema is only applicable if the client has no possibility to modify or even inspect the access envelope. Moreover abuse of a non-authorized user has to be blocked.

We can achieve this easily with a symmetric two key-pair encryption/signing policy. The catalogue service uses a private key to sign the envelope and encrypts it with the public key of the destination data server. The data server is the only receiver which can decrypt the envelope and validates the signature of the signing service with its public key.

We achieve with this basic schema the following:

- The client has no way modify what he is passing over as we requested. We have a secure authorization method.
- The data server can authorize a file access decoupled from any service. He needs only the public key of the signing catalogue service and its private key for decryption.
- TURLs and other storage information are send within the envelope and invisible for the client

III. A DATAMANAGEMENT FRAMEWORK WITH ACCESS ENVELOPE AUTHORIZATION

We have made a first implementation of an access envelope authorization framework using xrootd for data storage. As a catalogue service we were using the AliEn API service (see XXX) as a frontend to the AliEn2 file catalogue.

A. xrootd Server

xrootd daemon is a development of the Stanford University for SLAC. It's a file serving system which offers f.e.

- a distributed cache with load-balancing
- failover if files's are replicated during an I/O operation
- connection bundling (?!?!buendelung)
- asynchronous I/O

Its components are built with a plugin structure and the complete filesystem implementation can be exchanged. In xrootd the standard filesystem implementation is called OFS.

Authentication is also provided as a plugin architecture. At the date of this note however there was no release of authentication other than UID based available. GSI authentication is expected to be available soon.

For our purposes an authentication plugin could not be considered because it operates on a connection level and not on the file level. Therefore we provide for the authorization framework an inherited version of the OFS which adds the authorization functionality. Our EnvOFS is just an inherited class of the OFS which extends every every access function with the authorization framework.

The authorization envelope which is obtained from the catalogue service is appended to an URL as opaque information following the syntax:

```
URL : root : // < host - ip > : < port > // <
file - path > ?authz = < access - envelope >
&vo = < vo - name >
```

Since the xrootd client already supports opaque information, there are no changes necessary in the protocol necessary to move the authorization information from the client to the server. The URL definition requires only to provide the access envelope as a base64 encoded string.

In the OFS every file access function like open, stat, etc. gets two variables which are the file path and the opaque information. The EnvOFS functions read the authorization envelope, decode it with the key

mechanism corresponding to the defined VO and extract the authorization information from the envelope.

B. The Authorization Envelope Structure

The structure of the unencrypted envelope can be seen in table I.

```
-----BEGIN ENVELOPE-----
CREATOR:      Development-Catalogue V1.0
MD5:          b112c76d02b5832e8d83b74c102e0ca5
UNIXTIME:     1114497639
DATE:         Tue Apr 26 08:40:39 2005
EXPIRES:      0
EXPDATE:      never
CERTIFICATE   /CN=CERN/OU=.....
-----BEGIN ENVELOPE BODY-----
....
-----END ENVELOPE BODY-----
-----END ENVELOPE-----
```

TABLE I: Structure of an access envelope.

It is made up by two parts, an envelope header and the envelope body. The header contains f.e. the lifetime of an access envelope, the md5 sum to avoid manipulation and the certificate of the user who is allowed to use it. In the final implementation we will use GSI authenticated connections and every envelope will be cross-checked with the certificate subject given by the user connection.

The body which uses XML representation is shown in III B and provides the information, which file has to be accessed and in which way. An envelope can contain the authorization information for several file accesses. The client provides as the filepath in the URL the logical file name to be extracted from the envelope. The envelope can be seen as a small catalogue slice. It can be used for all files referenced in the envelope body. For every file entry in the envelope we specify the following XML tags:

- `lfn` - the logical filename in the file catalogue
- `turl` - the transport URL which defines the physical location of a file
- `access`
 - `read` - allows read access
 - `write-once` - allows to create a new not-existnat file with write access

- *write* - allows to write to a existing file
- *delete* - allows to delete an existing file
- *guid* - the unique file identifier envelope grants access
- *pturl* - the transport URL of an existing file to be opened for write mode
- *pguid* - the unique file identifier of an existing file to be opened for write mode

```

<authz>
  <file>
    <lfn>/vo/user/t/test/testfile.root</lfn>
    <turl>root://localhost:9999//tmp/file.root
    <access>write-once</access>
    <guid>d6efcb28-d53a-4a23-971e-7de279d3830e
    </guid>
  ....
</file>
<file>
  ....
</file>
</authz>

```

TABLE II: Example of the authorization envelope body

C. File Access Methods

1. read

For a *read* operation the authorization envelope defines the transport URL for a file and contains *access=read*.

The server uses the given TURL to open the file.

2. write-once

For a *write-once* operation the envelope contains *access=write-once* and the given TURL must be not existing, otherwise the server returns the error *permission denied*.

3. delete

The given TURL must be existing and the tag *access=delete* must be set. The file is deleted on the machine the client is connected to. The special case, where the client connects to a redirector will be explained later.

4. write

The *write* access to an existing file needs some special considerations. Since xrootd is a caching system which may contain replicas of a given file we produce a cache inconsistency if we modify with a write access one of the replicas.

Therefore we use a special naming scheme for logical file names in the catalogue and the physically stored files on the mass storage system. The AliEn2 file catalogue supports versioning of files. One logical filename is connected with a group of physical filenames which are referenced by the logical filename and a version number. Each physical filename has a different GUID assign to verify the physical difference of the file versions.

On the mass storage system we store files for the same LFN but different versions with physical filenames containing the GUID referenced in the file catalogue. This guarantees the uniqueness of filenames for different file versions.

If an existing file is opened in *write* mode, the EnvOFS open function creates a copy of the file referenced by *pturl* and *pguid* to a new version referenced by *turl* and *guid*. All following write operations modify the new copy.

By doing this we avoid cache inconsistencies for *write* operations on existing files.

D. Encoding of the Authorization Envelope

As previous explained we use a combination of signing and public key encryption. Since public key encryption would be slow for larger envelopes we are using a slightly different scheme following the same ideas.

The encryption of an authorization envelope is done in the following steps:

- Creation of the plain text envelope
- Creation of a 1024-bit (- something) random key (*cipher*)
- RSA Encryption of *cipher* with the private key of the catalogue service (

→

*cipher**)

- RSA Encryption of *cipher** with the public key of the destination xrootd (→*cipher***).
- Base64* Encoding of *cipher***
- CBC Encryption of the XML envelope body with the random key (→*envelope**)
- Base64* Encoding of *envelope**

The encoded cipher *cipher*** and envelope *envelope** are wrapped into a multi-line string as seen in table III D.

We use a modified base64 encoding where we replaced the `'/'` character by a `'_'` character. A problem arises if by chance a sequence with several `'/'` characters is generated in the base64 encoded string like f.e. `'//'` because `xrootd` replaces `'//'` in URLs with `'/'`. Moreover it is recommended not to use the `'/'` character in URLs to avoid confusion with file paths.

E. Decoding of the Authorization Envelope

The decoding reverses exactly the procedure described in the previous section:

- Base64* Decoding of *envelope**
- RSA Decryption of *cipher*** with the private key of the destination `xrootd` (\rightarrow *cipher**).
- RSA Decryption of *cipher** with the public key of the catalogue service (\rightarrow *cipher*)
- CBC Decryption of the XML envelope body with the random key *cipher* (\rightarrow *envelope*)
- Parsing of envelope header and body information

The coding and decoding of envelopes is implemented in a C++ class. A *swig* generated PERL is also available. For en-/decryption we use RSA and cipher routines of the OpenSSL library.

F. Security

The scheme uses at present a 1024 bit cipher to encode the information. The cipher is protected by the symmetric use of two private/public key pairs. Envelopes are issued only for a limited time period which is hidden inside the envelope. Ideally one should be able to deduce from the used cipher itself the expiration time. In this way one avoids the further use of a *hacked* cipher.

The modification of the envelope is prohibited by the included md5 sum.

The use of GSI authentication prohibits the abuse of stolen envelopes since the client has to authenticate with the certificate subject quoted in the envelope.

G. Performance

Benchmarks on a Pentium IV 3 GHz machine have shown, that the time for decoding a typical single

file access authorization envelope is in the order of 3-4ms, while the encoding time is faster by a factor of about 5.

The performance for encoding is not a real issue because there is no need to modify the cipher for every issued envelope. A change of the cipher for concurrent envelopes does not increase the security of the system. It is safe to change the cipher on a time scale of minutes, so the encoding needs only two parts, the fast CBC encryption and base64 encoding. More time critical is the authorization procedure (decoding time) on the server side. A single server can still perform several hundred file open operations per second.

A final judgement can be done if the GSI authentication has been integrated into this picture.

`xrootd` uses bundled connections if files are opened from the same client to the same server and they are kept alive as specified in the client/server configuration. Therefore there will be no additional time penalty after the first authentication for concurrent open operations from the same application.

H. Drawback for Load-balanced xrootd Cache Systems

`xrootd` allows to setup a load-balancing system where one machine is declared as the redirector for a group of normal `xrootd` file servers. All client requests are directed to the redirector. The redirector keeps a cache where a given file name can be found and redirects the client to connect to the appropriate `xrootd` server which keeps the file. If a file is not yet known the redirector broadcasts a location message to the `xrootd` servers. If one or more of the `xrootd` server keep the file the client is redirected to the machine with the lowest load.

In our authorization framework the redirector has to perform for every client request the decoding of the access envelope to know which file the client wants to read.

Already by construction the redirector is the most probable bottleneck of the system. On the other hand it is possible to use several redirectors and balance them with a DNS alias setup.

While tests with `xrootd` redirectors and without authorization have shown the possibility to manage in the order of 1000 files per seconds, a test using redirection and the authorization framework has still to be done.

Just by summing up the additional computing time one can expect one quarter of the referenced 1000 files per second.

If we don't want to hide the physical storage of files to clients, we can append the TURL unencoded to the envelope and run redirection without decoding. Only

server who finally access files needs to authorize.

I. Procedural Flow for Command Line Interaces

1. File Reading

The procedural flow for a read operation using the xrootd command line client *xrdcp* can be summarized as:

- Obtain authorization envelope from a Catalogue Service together with the URL of the xrootd to contact for a given LFN and the virtual organization name.
- Build access URL for xrootd in the following way: $URL = root : // < host > : < port > / < lfn > ? authz = < envelope > \&vo = < vo - name >$
- Execute copy command with the created URL

2. File Writing

- Obtain authorization envelope from a catalogue service together with the URL of the xrootd to contact for a given LFN and the virtual organization name.
- Build access URL like in the read case
- Execute copy command
- If successfully copied, execute the register function in the catalogue service with the used authorization envelope

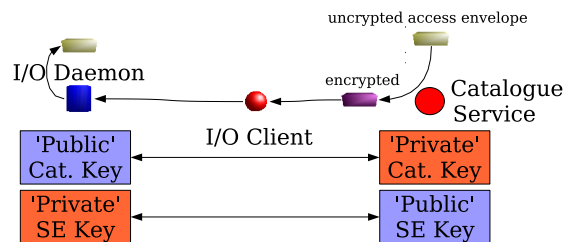
It gets more complicated if we want to hide completely the TURL referenced in the access envelope. After we copied a file to a xrootd server, the client cannot register the file in the file catalogue since he doesn't know the physical filename. We can only send the authorization envelope back to the catalogue service. The catalogue service must have also the private key of the xrootd to be able to read the envelope or must store in some reference DB the file information

for the registration of an used envelope. If we hand out the physical filename to the client, the client can perform a normal file registration command.

IV. CONCLUSIONS

Starting with the quoted requirements in GRID environments we have developed a highperformant and scalable framework for file access authorization with the maximum decoupling of the storage system

FIG. 2: Access Envelope Authorization Principle



from GRID services. The experience from the implementation within the AliEn GRID framework and xrootd is confirming this evidently.

One should make an effort for standardization of authorization with access envelopes to allow the same authorization scheme in any I/O protocol. It should be straight-forward to implement easily the same mechanism in other I/O servers like gridFTP.

With the described system we provide a intirely secure system. At the same time we have removed the mapping problem from virtual ownership in a virtual organization to the ownership and roles on the physical filesystem, where files are finally stored. Moreover we are able to deploy multi-VO file services. The deployment of data servers gets easier since the only requirement for this framework is the secure distribution of key-pairs for every supported virtual organization and the data server software itself.

[1] AliEn Grid Framework, <http://alien.cern.ch>

[2] P. Saiz, L. Aphecetche, P. Buncic, R. Piskac, J. -E. Revsbech and V. Segó, AliEn - ALICE environment on the GRID, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrom-

eters, Detectors and Associated Equipment, Volume 502, Issues 2-3, 21 April 2003, Pages 437-440

[3] The ROOT project, <http://root.cern.ch>

```

-----BEGIN SEALED CIPHER-----
ooM-omP6BG5dbzhY-S1bw3801HqdURORi0SRGNyGHWPx4BZUHPMm58DLAqJcigSvW73kdpWLMUM3
Am4aGGP5baEB9WJQpgi9YNwFCw24TybbcJmCQ31DbzJb-zfP3SB7PTmCrAgLdk7zS0tUKY2VPE0q
NGpDik0AzQUkymIRosc=
-----END SEALED CIPHER-----
-----BEGIN SEALED ENVELOPE-----
SqLE0h7AhDzo8CyzPsmJcDA38Z6LJ019v07hAvscwH8nMx6halWJr-gr7W25tiH15dZuII9vtW0B
FydDONRLhG5RGP3X+Vs9MobFP7AY+LTu8ZYFBaAODDCuUtMpEbvvtiz5C+qyndMwmwk8C7hGjVCR
QYkUBW5UDVZ1yXE-3N6XBJJiVJUhezr9BvrMw-QJnRQj2F1t19prfrkKE1T9aISmupRSfExsImJ0
Rsc69-0mRRnUwkOLLR+ffnm7zap4EonYgBCgeQ+e1ouujq+zzyMY83uSGJJ5dJTE8n51ACDDr+jV4
ek0swLt21p6MhbTehmDiXd9iN1VGrw4JnUIKSsGfvLxDgJeaDWhGzWp9q798V0c5McvxlzZCAzp
SPzBbxMNg0xLyD3xjsRap53k0kR0cttekPVyylw6zExhCuv0D7mX3vjd3w==
-----END SEALED ENVELOPE-----

```

TABLE III: Structure of an encoded access envelope.