Certificate Repository Manual

Robert Olson

4th August 2003

olson@mcs.anl.gov

Abstract

A Certificate Repository manages a collection of certificates on behalf of a user.

Contents

1 Introduction

2 Certificate Management

In [the cert mgmt document] we discuss the requirements of the certificate management infrastructure in the AG. In brief, the AG software expects there to be two “pools” of certificates: a set of user identity certificates, from which Globus proxy certificates can be created; and a set of trusted CA certificates.

The usual operations on these pools of certificates are the lookup and creation of proxies from the identity certificates. Less often, identity certificates will be imported and exported. Even less often, trusted CA certificates will be added and removed.

An AG certificate manager will use a single certificate repository for both the identity and trusted CA certificates. The two certificate types are distinguished using the metadata defined for each certificate. The metadata key *AG.CertType* will have the value *identity* for identity certs, and *ca* for trusted CA certs.

Since the Globus toolkit requires the trusted CA certs to reside in a flat directory with a defined naming structure (that is different than that used by the certificate repository), the Globus trusted CA directory will be created from the certificate repository each time a trusted CA certificate is added to or removed from the repository.

3 Requirements

The certificate repository component of the certificate management package provides mechanism for an application to manage trusted CA and user or service identity certificates, as well as supporting the requesting of new certificates and the creation of Globus proxy certificates.

The design philosophy behind the certificate repository is that it should be considered to be a fairly low-level tool.

This module defines class `CertificateDescriptor` to represent individual certificates. This wrapper class does not provide an independent representation for the certificate: the fundamental description of the certificate is the stored form of the certificate in the filesystem. Internally, the pyOpenSSL module is used to read the certificate from the filesystem and cache OpenSSL data structures which describe the certificate; pyOpenSSL is then used to manipulate these data structures to read the certificate.

4 Repository Structure

A certificate repository has the following structure in the filesystem:

```
<repo_root>/
            metadata.db
            certificates/<subject_hash>/
                                 <issuer_serial_hash>/cert.pem
                                                      user_files/
            requests/<modulus_hash>.pem
            privatekeys/<modulus_hash>.pem
```

All certificates and requests for a given subject are stored in the directory `<subject_hash>`, where `<subject_hash>` is the hex version of the MD5 digest of the DER form of the Subject of the certificate. The actual certificateis stored in a subdirectory `<issuer_serial_hash>`; this name is the hex version of the MD5 digest of the DER form of the issuer’s distinguished name concatenated with the serial number of the certificate. (Together, `<subject_hash>` and `<issuer_serial_hash>` uniquely identify a certificate). If desired, the user of the repository can store files in a subdirectory `user_files` of the certificate's directory.

Metadata for the repository is stored in a Berkeley database named `metadata.db` at the toplevel of the repository hierarchy. Metadata for particular certificate and metadata key `<metadata_key>` is stored using this key into the metadata:

```
cert|<subject_hash>|<issuer_serial_hash>|<metadata_key>
```

Metadata for a particular certificate request is stored here:

```
req|<subject_hash>|<modulus_hash>|<metadata_key>
```

If a certificate request is pending for a given subject, the request document is stored in the file `<modulus_hash>.req.pem`, where `<modulus_hash>` is the hex version of the MD5 hash of the modulus of the key used to sign certificate request.

The private keys for any certificate or certificate request are found in the `privatekeys` directory, in files named `<modulus_hash>` as defined above.

If a certificate has a valid Globus proxy certificate in existence, the proxy will be stored in the `<subject_hash>/<issuer_serial_hash>` directory; they cannot be stored independently because the proxy certificates do not have uniquely issued serial numbers.

Certificate metadata is stored using a Berkeley DB file. The metadata is for the use of applications of the certificate store; as such, the namespace in the database must be managed. We define a simple hierarchical structure for this naming. Database keys are sequences of strings with dots separating the layers in the hierarchy. The first portion of the name is the application name; the rest of the hierarchy is defined by the application itself. The application name `System` is reserved for the certificate management software itself.

5 Certificate Management Operations

We discuss various operations on a certificate repository and their effects on the repository.

5.1 Importing a new certificate

Certificates are added to the repository through the certificate import mechanism. Importation of a certificate follows the following steps:

- The file containing the new certificate is identified, and the type of the file confirmed (PEM, PKCS12, raw DER, etc).
- We determine if the certificate is an identity certificate (in which case it requires a private key as well, which can either be included in the certificate file or specified as a separate file) or a trusted CA certificate.
- The format of the certificate is translated, if necessary, to be understood by the certificate management software.
- The certificate is loaded into a pyOpenSSL certificate object. Verification of the correctness of the certificate is performed.
- The repository is checked to determine whether the certificate is already present. If not, the directory hierarchy for the certificate is created and the certificate written to it.

If there is a private key, the private key is copied to the appropriate location as well.

The metadata for the certificate is initialized. The following entries are created:

- System.importDate
- System.certType
- System.originalSource

5.2 Exporting a certificate

A user can export a certificate for use in other applications.

5.3 Browsing certificates

The certificate repository itself does not support browsing; however, through its query mechanisms it can return to the application the sets of certificates matching the appropriate critera for the browser.

5.4 Creating Globus proxy certificates

5.5 Default certificates

6 Using the repository

Applications make use of the certificate repository through its API. The primary interface is via the `CertificateRepository` class; at application startup time the application creates an instance of the `CertificateRepository` class, passing to the constructor the directory in which the repository should be found. Unless the keyword argument `create = 1` is passed to the constructor, the repository must already exist. If it does not exist, invocation of the constructor will raise the `RepositoryNotPresent` exception. If the repository directory is present, but somehow corrupted (or does not actually contain a certificate respository), the constructor will raise the `RepositoryCorrupt` exception.

Note: The distinction between creating a new repository and opening an existing repository is designed in part to aid in the bootstrapping problem wherein when a new certificate repository is to be created we want to perform an initial import of certain certificates (e.g. the user's Globus identity certificate, the trusted CA certificates from the default Globus location).

An individual certificate is described by an instance of the `CertificateDescriptor` class. A certificate descriptor provides methods to allow the user to query the certificate for its attributes (issuer, subject, etc), and to use as a target for per-certificate operations in the repository.

For instance, to create a Globus proxy for a given DN, a user might write the following:

```
repo = CertificateRepository("/home/user/.repo")
descList = repo.FindCertificates("issuer",  "O/...")
repo.CreateGlobusProxy(descList[0])
```

The `CertificateRepository` class defines a number of lookup methods for finding certificates. These include

`FindCertificate()` Find the certificate with the given subject, issuer, and serial number.

`FindCertificatesWith<Attr>()` Returns all certificates which have the given <Attr> (subject, issuer, etc).

`FindCertificatesWithMetadata()` Returns all certificates with metadata matching the given value.

`FindCertificates()` Returns all certificates for which the given evaluation function returns true.

6.1 Creating certificate requests

The certificate repository provides the mechanism for easy creation of certificate requests. The application is responsible for gathering the name/value pairs that make up the subject name of the request. For instance, to create a request named “O=Grid, OU=Access Grid, OU = ANL, CN=Bob Olson” one might write the following:

```
nlist = [("O", "Grid"),("OU", "Access Grid"),
         ("OU", "ANL"),("CN", "Bob Olson")]
cdesc = repo.CreateCertificateRequest(nlist, "secretPhrase")
# Now to write to a file to be mailed
reqText = cdesc.ExportPEM()
fh = open(‘‘out.pem’’, ‘‘w’’)
fh.write(reqText)
fh.close()
```

7 `AccessGrid.CertificateManager` — Python certificate management tools

class `CertificateRepository`(*directory*)
A `CertificateRepository` instance holds a set of X.509 certificates, certificate requests, private keys, and Globus signing policy definition files. It is instantiated with the directory in which these items are stored.

class `CertificateDescriptor`()
A `CertificateDescriptor` instance describes a single certificate resident in a repository. Instances of this class are not created by the application; rather, they are returned as a result of an `Import` operation on a repository or from a query method.

class `CertificateRequestDescriptor`()
A `CertificateDescriptor` instance describes a certificate request as kept in a repository. Instances of this class are not created by the application.

7.1 CertificateRepository Objects

The `CertificateRepository` class defines the following methods:

`FindCertificatesWithSubject`(*str*)
Retrieve all certificates that have a subject name of *str*.

`FindCertificatesWithIssuer`(*str*)
Retrieve all certificates that have a issuer name of *str*.

`FindCertificatesWithMetadata`(*mdKey, mdValue*)
Retrieve the certificates matching the given *dn*.

`FindCertificates`(*predicate*)
Invoke *predicate* on each certificate in the repository. Return the list of certificates for which *predicate(cert)* returns true.

`GetAllCertificates``()`
Retrieve a list of all certificates.

`ImportCertificatePEM``(`*cert, key = None*`)`
Import a PEM-formatted certificate from file *cert*. If *key* is not None, *key* is the private key file for *cert*.

`ImportCertificateDER``(`*cert, key = None*`)`
Import a DER-formatted certificate from file *cert*. If *key* is not None, *key* is the private key file for *cert*.

`CreateCertificateRequest``(`*nameEntries, passphrase, keyType = KEYTYPE_RSA, bits = 1024, messageDigest = "md5", extensions = None*`)`
Create a new certificate request.

nameEntries is a list of 2-tuples `(key, value)` where key is one of the standard distinguished name keys:

CN Common name

C Country name

L Locality name

ST State or province name

O Organization name

OU Organizational unit name

emailAddress Email address

If *passphrase* is a string, it is used as the passphrase for the private key of the certificate request. If it is a Python callable object, the string value that is returned from its invocation is used as the passphrase for the private key. If *passphrase* is `None`, the private key will not be encrypted.

The *keyType* argument defines the type of key to be generated. Valid values are `KEYTYPE_RSA` and `KEYTYPE_DSA`.

The *bits* defines the size of the key.

The *messageDigest* argument defines the message digest to be used in signing the certificate request.

The *extensions* argument defines the X509 extensions to be used in teh creation of the certificate request. This argument must be a list of 3-tuples *(name, critical_flag, value)*. The *name* field specifies the name of the extension. The *critical* field should be 1 if the extension is critical, 0 otherwise. The *value* field is the value for the extension.

If the *extensions* argument is `None`, the following extensions list will be assumed:

```
[
    ("nsCertType", 0, "client,server,objsign,email"),
    ("basicConstraints", 1, "CA:false")
]
```

This method returns a `CertificateRequestDescriptor` instance describing the new request.

7.2 CertificateDescriptor Objects

The `CertificateDescriptor` class defines the following methods:

`GetIssuer``()`
Return the issuer of the certificate as a pyOpenSSL X509Name object.

`GetSubject``()`
Return the subject of the certificate as a pyOpenSSL X509Name object.

`IsValid``()`
Returns true if the certificate is valid.

`GetMetadata``(`*key*`)`
: Return the metadata associated with *key*.

`SetMetadata``(`*key, value*`)`
: Set the metadata associated with *key* to be *value*.

`GetFilePath``(`*filename*`)`
: Determine the pathname the application should use in associating the given *filename* with this certificate.

7.3 CertificateRequestDescriptor Objects

The `CertificateRequestDescriptor` class defines the following methods:

`GetSubject``()`
: Returns the subject name in this certificate request.

`GetModulus``()`
: Returns the public-key modulus of this certifidate request.

`GetModulusHash``()`
: Returns the MD5 has of the public-key modululs of this certifidate request.

Module Index

Index