GSI protocol specifications

Purpose of the document

This document describes the XRootD implementation of the GSI protocol [1][2]; the XRootD protocol is described in [3].

The protocol version described in the document corresponds at the head of the GIT master branch at the time of writing, which is supposed to go in v4.9 (internal version of the GSI protocol 10400). This includes the verification of the server identity, as explained in the related section. Notable changes with respect versions prior to v4.9 (internal version lesser or equal to 10300) are described in Appendix C.

Related documents

The cryptographic functions used by the GSI protocol implementation are provided by XrdCrypto [4]. A set of utilities used in common with the PWD authentication modules is provided by XrdSut [5].

Versions

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>26 August 2018</td>
<td>Created; Data structures; protocol interface</td>
</tr>
<tr>
<td>0.1</td>
<td>16 October 2018</td>
<td>Client handshake steps</td>
</tr>
<tr>
<td>0.2</td>
<td>17 October 2018</td>
<td>Server handshake steps</td>
</tr>
<tr>
<td>0.3</td>
<td>22 October 2018</td>
<td>More about delegation</td>
</tr>
<tr>
<td>0.4</td>
<td>25 October 2018</td>
<td>More about delegation; add sketch of delegation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Chain</td>
</tr>
<tr>
<td>0.5</td>
<td>21 December 2018</td>
<td>Document XrdSutBuffer / XrdSutBucket serialization, introduction of DH parameters signing, and IV enabling</td>
</tr>
<tr>
<td>Purpose of the document</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>Related documents</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Versions</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Authentication protocol interface</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Data structures</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Buffers and Buckets</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>XrdSutBuffer serialization</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Bucket types</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>Class members of XrdSecProtocolgsi</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Ciphers</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Delegation options</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Handshake</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>Description</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>Implementation dissection</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Class gsiHSVars</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Global and Main buffers</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>Steps</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>Client side</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>Common pre-step processing</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>Step processing</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>Step: kXGS_init</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>Received buffer</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>Additional input information</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>Actions performed</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>Preparation of the reply to kXGS_init</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>Step: kXGS_cert</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>Received buffer</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>Actions performed</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>Preparation of the reply to kXGS_cert</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>Step: kXGS_pxyreq</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>Received buffer</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>Actions performed</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>Preparation of the reply to kXGS_pxyreq</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>Common post-step processing</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>Errors / failures</td>
<td>17</td>
<td></td>
</tr>
</tbody>
</table>

XRootD GSI Protocol Specifications 2
Server side 18
  Common pre-step processing 18
  Step processing 18
    Step: kXGC_certreq 18
    Received buffer 18
    Actions performed 19
    Preparation of the reply to kXGC_certreq 19
    Step: kXGC_cert 19
    Received buffer 19
    Actions performed 20
    Preparation of the reply to kXGC_cert 21
    Step: kXGC_sigpxy 21
    Received buffer 21
    Actions performed 22
    Preparation of the reply to kXGC_sigpxy 22
  Common post-step processing 22
  Errors / failures 22

(Delegated) Proxy certificates 23

Server identity verification 24

References 25

Appendix A - Relevant parts of header files 26
  XrdSecEntity.hh 26
  XrdSecInterface.hh 27
  XrdSutBucket.hh 31
  XrdSutBuckList.hh 32
  XrdSutBuffer.hh 33
  XrdSutAux.hh (excerpt) 34
  XrdSecProtocolgsi.hh 35

Appendix B - Details of bucket types 45

Appendix C - Versions prior to 10400 / v4.9 47
  Server host name verification 47
  Transmission of DH parameters 47
    Client: step kXGS_cert processing 47
    Received buffer 47
    Actions performed 47
    Preparation of the reply to kXGS_cert 48

XRootD GSI Protocol Specifications 3
Server: step kXGC_cert processing
Received buffer
Actions performed
Authentication protocol interface

Authentication protocols in XRootD are provided as plug-ins. Protocols are identified by a string of max \texttt{XrdSecPROTOOIDSIZE-1} characters. \texttt{XrdSecPROTOOIDSIZE} is defined in \texttt{XrdSec/XrdSecEntity.hh} and its value is 8. The ID for the GSI protocol is \texttt{gsi}.

The authentication protocol plug-in must provide a concrete implementation of the \texttt{XrdSecProtocol}, defined in \texttt{XrdSec/XrdSecInterface.hh}, in addition to C functions to load and initialize the protocol.

Once the protocol is loaded and initialized, clients will call the method

\begin{verbatim}
virtual XrdSecCredentials *getCredentials(XrdSecParameters *parm=0, 
XrdOucErrInfo *einfo=0)=0;
\end{verbatim}

to get a buffer of information to be sent to the server, and servers will call

\begin{verbatim}
virtual int Authenticate (XrdSecCredentials *cred, 
XrdSecParameters **parms, 
XrdOucErrInfo *einfo=0)
\end{verbatim}

on the buffer of information received by the client. The application using the authentication framework and mediating the exchange, has no knowledge of the content of the buffers exchanged, which are produced and analyzed inside the plug-in code.

Data structures

XRootD authentication handshakes use a generic data structure to exchange information between the two parties, client and server. The generic structure, called \texttt{XrdSecBuffer}, is defined in \texttt{XrdSec/XrdSecInterface.hh}. It contains a generic buffer and its size.

The structure \texttt{XrdSecCredentials}, defined in the same file, is typedef from \texttt{XrdSecBuffer} and assumes that the first \texttt{XrdSecPROTOOIDSIZE-1} characters of the buffer contain the protocol name (or ID).
Buffers and Buckets

Internally, XrdSec protocol implementations organize the buffer according to needs. The gsi protocol interprets the buffer as a serialization of the class XrdSutBuffer, defined in XrdSut/XrdSutBuffer.hh. The buffer is further organised in buckets (class XrdSutBucket; see XrdSut/XrdSutBucket.hh), which contain the information to be processed.

The members of XrdSutBuffer are, in the order:

- XrdSutBuckList fBuckets // list of buckets
- XrdOucString fOptions // string with options
- XrdOucString fProtocol // string with the protocol name or ID
- kXR_int32 fStep // 32 bit integer with a counter indicating the step of the handshake

The XrdSutBuckList class implements a light single-linked list to store and navigate through buckets. The XrdSutBucket structure contains:

- kXR_int32 type // 32 bit integer with the type of the bucket
- kXR_int32 size // 32 bit integer with the size in bytes
- char* buffer // the content of the bucket

XrdSutBuffer serialization

The content of XrdSutBuffer is serialized into a buffer of length

\[
\text{fProtocol\_length + 1 + 2} \cdot S_{32} + N_{\text{buckets}} \cdot 2 \cdot S_{32} + \text{Sum\_of\_bucket\_sizes}
\]

(number in bytes). Here \( S_{32} \) is \( \text{sizeof(kXR\_int32)}=4 \), and \( N_{\text{buckets}} \) is the number of bucket in the XrdSutBuckList list.

The buffer contains, in order:

1. \( \text{fProtocol\_length + 1} \) bytes with the protocol ID; max XrdSecPROTOIDSIZE; this is interpreted as a string;
2. \( S_{32} \) bytes with the step number, marshalled;
3. For each bucket:
   a. \( S_{32} \) bytes with the bucket type, marshalled;
   b. \( S_{32} \) bytes with the bucket size, marshalled;
   c. the content of the bucket.
4. \( S_{32} \) bytes with the termination type KXRS_none, marshalled;
The composition of the buffer is shown graphically in Figure 1.

Figure 1. XrdSutBuffer serialized

<table>
<thead>
<tr>
<th>Protocol ID, null terminated string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step number, 32-bit integer, marshalled</td>
</tr>
<tr>
<td>Type of bucket 1, 32-bit integer, marshalled</td>
</tr>
<tr>
<td>Size of bucket 1, 32-bit integer, marshalled</td>
</tr>
<tr>
<td>Content of bucket 1</td>
</tr>
<tr>
<td>Type of bucket 2, 32-bit integer, marshalled</td>
</tr>
<tr>
<td>Size of bucket 2, 32-bit integer, marshalled</td>
</tr>
<tr>
<td>Content of bucket 2</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Termination type kXSR_none, 32-bit integer, marshalled</td>
</tr>
</tbody>
</table>

Bucket types

Bucket types are given by the `enum kXRSBucketTypes` in `XrdSut/XrdSutAux.hh` and reported in Table 1 in Appendix B.
Class members of \texttt{XrdSecProtocolgsi}

The class \texttt{XrdSecProtocolgsi} (\texttt{XrdSecgsi/XrdSecProtocolgsi.hh}) has the following members:

- \texttt{int \ options}
- \texttt{XrdCryptoFactory *sessionCF}
- \texttt{XrdCryptoCipher *sessionKey}
- \texttt{XrdSutBucket *bucketKey}
- \texttt{XrdCryptoMsgDigest *sessionMD}
- \texttt{XrdCryptoRSA *sessionKsig}
- \texttt{XrdCryptoRSA *sessionKver}
- \texttt{X509Chain *proxyChain}
- \texttt{bool \ srvMode}
- \texttt{gsiHSVars *hs}

**Handshake**

- Chosen crypto factory
- Session Cipher, as result of the handshake
- Bucket with the key in export form
- Message Digest (unused during handshake)
- RSA key to sign
- RSA key to verify
- Chain with the delegated proxy on servers
- TRUE if server mode
- Temporary handshake information

**Ciphers**

The shared cipher is generated using the Diffie-Hellman key agreement method [6]. Default ciphers, in order of preference, are:

\texttt{aes-128-cbc bf-cbc des-ede3-cbc}.

An initialization vector (IV) of 16 bytes (OpenSSL constant \texttt{EVP_MAX_IV_LENGTH}) is generated for each encryption and prepended to the encrypted buffer. The length of the IV is communicated by the client to the server with the name of the chosen cipher (see relevant section).

**Delegation options**

The proxy delegation options are controlled internally by the settings saved in the \texttt{Options} field of the \texttt{gsiHSVars} instance attached to the protocol. The \texttt{enum kgsiHandshakeOpts} in \texttt{XrdSecgsi/XrdSecProtocolgsi.hh} defines the meaning of the bits, reported in Table 2.
Table 2. Settings controlling proxy delegation

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Set by</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>kOptsDlgPxy</td>
<td>1</td>
<td>S</td>
<td>Ask for a delegated proxy</td>
</tr>
<tr>
<td>kOptsFwdPxy</td>
<td>2</td>
<td>C</td>
<td>Forward local proxy</td>
</tr>
<tr>
<td>kOptsSigReq</td>
<td>4</td>
<td>C</td>
<td>Accept to sign delegated proxy</td>
</tr>
<tr>
<td>kOptsSrvReq</td>
<td>8</td>
<td>S</td>
<td>Server request for delegated proxy</td>
</tr>
<tr>
<td>kOptsPxFile</td>
<td>16</td>
<td>S</td>
<td>Save delegated proxies in file</td>
</tr>
<tr>
<td>kOptsPxCred</td>
<td>64</td>
<td>S</td>
<td>Save delegated proxies as credentials</td>
</tr>
</tbody>
</table>

Handshake

Description

The authentication handshake is part of the login process. It is initiated by the server when configured to require strong authentication.

For gsi the goal of the handshake is to mutually verify the credentials - the server verifies the client proxy certificate, the client verifies the server certificate, and to create a shared secret to encrypt the rest of the handshake and further communication.

Optionally, after a successful handshake, a delegate client proxy certificate can be produced to enable further authentication handshakes initiated by the server on behalf of the client, for example in the case of a Third Party Copy.

According to the protocol, a delegate client proxy certificate is a proxy certificate generated by the server using as a base the client proxy certificate, and then signed by the client using the private key of its proxy. The full chain for a delegated client proxy certificate is, therefore,
To overcome problems with the early versions of OpenSSL, the XRootD gsi implementation supported the option to recreate the full client proxy on the server side by sending over the private key of the client proxy certificate; while still supported, this is to be considered deprecated.

Implementation dissection

The authentication handshake is started by the server and continues until a consensus is reached, failure or success. In the case of a successful handshake, the relevant information about the authenticated client is saved in a instance of the XrdSecEntity class owned by the XrdSecProtocol instance.

Class gsiHSVars

During the handshake, both parts keep the relevant state of the handshake in a instance of the class gsiHSVars, defined in XrdSecgsi/XrdSecProtocolgsi.hh. The class gsiHSVars contains the following members:

- **int Iter**: Iteration number
- **time_t TimeStamp**: Time of last call
- **XrdOucString CryptoMod**: Crypto module in use
- **int RemVers**: Version run by remote counterpart
- **XrdCryptoCipher *Rcip**: Reference cipher
- **XrdOucString *Cbck**: Bucket with the certificate in export form
- **XrdSutBucket ID**: Handshake ID (dummy for clients)
- **XrdSutPFEntry *Cref**: Cache reference
- **XrdSutPFEntry *Pent**: Pointer to relevant file entry
- **X509Chain *Chain**: Chain to be eventually verified
- **XrdCryptoX509Crl *Crl**: Pointer to CRL, if required
- **X509Chain *PxyChain**: Proxy Chain on clients
- **bool RtagOK**: Rndm tag checked / not checked
- **bool Tty**: Terminal attached / not attached
- **Int LastStep**: Step required at previous iteration
- **int Options**: Handshake options;
- **int HashAlg**: Hash algorithm of peer hash name;
Global and Main buffers

The message exchanged between client and server corresponds to the serialization of a global buffer, internal name $bpar$. The buffer $bpar$ contains control/auxiliary information and a buffer with the main information of the handshake serialized (internal name $bmai$).

The control/auxiliary information consists in: protocol version number; list of cryptographic modules; hash of the client certificate issuer; client options (delegation).

Steps

The handshake consists in a set of steps. In the implementation the steps are described by dedicated enum variables, defined in `XrdSecgsi/XrdSecProtocolgsi.hh`.

The client steps are enumerated by the enum `kgsiClientSteps`. They describe the handshake steps from the client point of view, and are encoded in the messages send by the client to the server. They are reported in Table 3.

<table>
<thead>
<tr>
<th>Client step</th>
<th>Code</th>
<th>Description</th>
<th>Rtag</th>
<th>Encrypted</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXGC_none</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>kXGC_certreq</td>
<td>1000</td>
<td>Request server certificate</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>kXGC_cert</td>
<td>1001</td>
<td>Packet with client (proxy) certificate</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>kXGC_sigpxy</td>
<td>1002</td>
<td>Packet with signed client proxy certificate</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

The server steps are enumerated by the enum `kgsiServerSteps`. They describe the handshake steps from the server point of view, and are encoded in the messages send by the server to the client. They are reported in Table 4.
Table 4. Server steps enum types as defined in XrdSecProtocolgsi.hh. The Rtag column indicates if an Rtag, signed with the client private key, is present. The last column indicates whether the main buffer bmai is encrypted with the session cipher.

<table>
<thead>
<tr>
<th>Server step</th>
<th>Code</th>
<th>Description</th>
<th>Rtag</th>
<th>Encrypted</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXGS_none</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>kXGS_init</td>
<td>2000</td>
<td>fake code used the first time</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>kXGS_cert</td>
<td>2001</td>
<td>packet with server certificate</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>kXGS_pxyreq</td>
<td>2002</td>
<td>packet with client proxy request to be signed</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

Client side

Common pre-step processing

The information exchanged is first deserialized and then interpreted. The following steps are performed by the client on the buffer received by the server:

1. Update the TimeStamp and the internal counter Iter in gsiHSVars;
2. Deserialize the received buffer (internal name bpar);
3. Check the protocol ID string to be “gsi”;
4. Determine the step required by the server
   a. If bpar->GetStep() is null, assume it is kXGS_init
5. Make sure that XrdSecEntity::name if filled for the protocol instance; honour the env XrdSecUSER settings if needed.

The remaining analysis of the received buckets depends on the server step.

Step processing

Step: kXGS_init

Received buffer

The received buffer contains the protocol initialization string:

"v:<version>,c:<crypto module>,ca:<hash of server CA>"
where:

<version> protocol version run by the server int
<crypto module> pipe '|' separate list of crypto modules string
<hash of server CA> pipe '|' separated hashes for the server CA string

Additional input information

The client also honours possible settings via the login URL. The following variables are checked:

xrd.gsiusrpxy location of the user proxy UsrProxy
xrd.gsiusrcrt location of the user certificate UsrCert
xrd.gsiusrkey location of the user certificate key UsrKey

The last column indicates the name of the internal variable overwritten by the corresponding URL setting.

Actions performed

The client performs the following actions:

1. Parse the protocol initialization string and saves the extracted information in the internal handshake state structure;
2. Resolve, if any, the place-holders in user certificate, key and proxy file paths (UsrCert, UsrKey and UsrProxy, respectively)
3. Loads the local proxy certificate from /tmp/x509up_u<uid> or the path defined by the env X509_USER_PROXY
   a. If no valid proxies are found, initialize the proxy using the end-user certificate from $HOME/.globus/usercert.pem or the path defined by the env X509_USER_CERT
      i. If needed - and the process is attached to a TTY - the password for the end-user certificate private key will be prompted; the private key is taken from $HOME/.globus/userkey.pem or X509_USER_KEY.
   b. Saved in the local gsiHSVars state variable for optimized subsequent use
4. Loads the RSA private key of the proxy is loaded in sessionKsig and used for signatures.

Preparation of the reply to kXGS_init

The client creates the main handshake information buffer bmai as a copy of the received global buffer bpar.

The following information is added the global buffer bpar:

1. A bucket of type kXRS_cryptomod with the name of the chosen cryptographic module; ssl is the only one available currently;
2. A bucket of type kXRS_version with the client version, 32-bit int, marshalled;
3. A bucket of type kXRS_issuer_hash with the hash of the issuer of the user certificate
a. For compatibility arguments, more than one hash can be given; these must be separated by a ‘\|’.

4. A bucket of type \texttt{kXRS\_clnt\_opts} with the client options as defined by the \texttt{Options} field of the client \texttt{gsiHSSVars} instance, a 32-bit integer, marshalled.

The client sets the next step, internally \texttt{nextstep}, for the server to be \texttt{kXGC\_certreq}.

Step: \texttt{kXGS\_cert}

\textit{Received buffer}

The information is contained in both the global and main buffers as described in Tables 5a and 5b.

Table 5a. Content of the global buffer bpar for step \texttt{kXGS\_cert}

<table>
<thead>
<tr>
<th>\textit{Bucket Type}</th>
<th>\textit{Bucket content}</th>
<th>\textit{Example, comments}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{kXRS_cryptomod}</td>
<td>\texttt{const char *}</td>
<td>\texttt{ssl}</td>
</tr>
<tr>
<td>\texttt{kXRS_cipher_alg}</td>
<td>\texttt{const char *}</td>
<td>\texttt{aes-128-cbc:bf-cbc:des-ede3-cbc}</td>
</tr>
<tr>
<td>\texttt{kXRS_cipher}</td>
<td>Raw buffer</td>
<td>DH parameters, signed</td>
</tr>
<tr>
<td>\texttt{kXRS_md_alg}</td>
<td>\texttt{const char *}</td>
<td>\texttt{sha256:sha1}</td>
</tr>
<tr>
<td>\texttt{kXRS_x509}</td>
<td>\texttt{const char *}</td>
<td>Server certificate, PEM format</td>
</tr>
<tr>
<td>\texttt{kXRS_main}</td>
<td>\texttt{const char *}</td>
<td>\texttt{bmai} (see Table 5b), plain text</td>
</tr>
</tbody>
</table>

Table 5b. Content of the global buffer bmai for step \texttt{kXGS\_cert}

<table>
<thead>
<tr>
<th>\textit{Bucket Type}</th>
<th>\textit{Bucket content}</th>
<th>\textit{Example, comments}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{kXRS_signed_rtag}</td>
<td>Raw buffer</td>
<td>Client challenge signed</td>
</tr>
<tr>
<td>\texttt{kXRS_rtag}</td>
<td>\texttt{const char *}</td>
<td>Server challenge</td>
</tr>
</tbody>
</table>

Actions performed

The client performs the following actions:

1. Check the cached timestamp against the current timestamp; allow for 300 seconds skew;

2. Get from the global buffer the bucket of type \texttt{kXRS\_cipher\_alg} with the cipher algorithm list supported by the server; chosen the first one supported locally; update the
bucket with the name of the chosen algorithm; the length of the IV which will prefix encrypted buffers is passed as '<#<IV_length>>', for example: aes-128-cbc#16
3. Get from the global buffer the bucket of type kXRS_x509 with the server certificate and used it to finalize the server certificate chain; verify the chain validity.
   a. Drop bucket kXRS_x509 from the global buffer;
4. Verify the server identity: check the server hostname against the certificate Distinguished Name (DN) and, possibly, the Alternative names;
5. Extract the public key from the server certificate and save it in sessionKver;
6. Get from the global buffer the bucket of type kXRS_cipher with the server public parameters for DH key agreement, initialize the session cipher and store it in sessionKey;
   a. Drop bucket kXRS_cipher from the global buffer;
7. Get from the global buffer the bucket of type kXRS_md_alg with the message digest algorithm list supported by the server; chosen the first one supported locally; update the bucket with the name of the chosen algorithm
8. Get from the global buffer the bucket of type kXRS_main and deserialize it .

Preparation of the reply to kXGS_cert

The following information is added the global buffer bpar:

1. A bucket of type kXRS_puk with the client public key, extracted from the client certificate and exported into a string in PEM format, i.e. base64 encoded data surrounded by header lines.
2. A bucket of type kXRS_cipher with the client public parameters for DH key agreement, signed with the client private key;
3. A bucket of type kXRS_x509 with the client proxy certificate;

The following information is added the global buffer bmai:

1. A bucket of type kXRS_user with the name of the user.

The client sets the next step, internally nextstep, for the server to be kXGC_cert.

Step: kXGS_pxyreq

Received buffer

The information is contained in both the global and main buffers as described in Tables 6a and 6b.

Table 6a. Content of the global buffer bpar for step kXGS_pxyreq

<table>
<thead>
<tr>
<th>Bucket Type</th>
<th>Bucket content</th>
<th>Example, comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXRS_cryptomod</td>
<td>const char *</td>
<td>ssl</td>
</tr>
<tr>
<td>kXRS_main</td>
<td>Raw buffer</td>
<td>bmai encrypted with sessionKey</td>
</tr>
</tbody>
</table>
Table 6b. Content of the global buffer bmai for step kXGS_pxyreq

<table>
<thead>
<tr>
<th>Bucket Type</th>
<th>Bucket content</th>
<th>Example, comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXRS_x509_req</td>
<td>const char *</td>
<td>Proxy request, PEM format</td>
</tr>
<tr>
<td>kXRS_signed_rtag</td>
<td>Raw buffer</td>
<td>Client challenge signed</td>
</tr>
</tbody>
</table>

Actions performed

The client performs the following actions:

1. Check the cached timestamp against the current timestamp; allow for 300 seconds skew;
2. Get from the global buffer the bucket of type kXRS_main, decrypt with sessionKey and deserialize it
3. If delegation option kOptsSigReq is set
   a. Get from the main buffer the bucket of type kXRS_x509_req with the proxy request; extract the request into a XrdCryptoX509Req instance
   b. Sign the request with the client proxy private key
   c. Export the signed request into a bucket of type kXRS_x509; add the bucket to the main buffer.
4. Else, if delegation option kOptsFwdPxy is set
   a. Export the private key of the client proxy as string; add it to the main buffer as bucket of type kXRS_x509;

Preparation of the reply to kXGS_pxyreq

The client sets the next step, internally nextstep, for the server to be kXGC_sigpxy. In case of errors in the processing of the proxy request, a bucket with the error message is added the global buffer bpar.

Common post-step processing

The following actions are performed after the processing of the step peculiarities:

1. The step nextstep is set both in the global buffer and in the main buffer;
2. If a random challenge was present in the received main buffer, in the form of a bucket of type kXRS_rtag, sign the challenge with the private key sessionKsig; the bucket type is updated to kXRS_signed_rtag;
3. A new random challenge is added to the main bucket as a bucket of type kXRS_rtag;
4. The new random challenge and the current time stamp are saved to a local cache;
5. The main buffer is serialized; the result of the serialization is used to update - or add - a bucket of type \texttt{kXRS\_main} into the global buffer.

6. The main bucket is encrypted with session cipher \texttt{sessionKey}
   a. This does not apply to the first client step when \texttt{sessionKey} is not yet defined

7. The global buffer is serialized; a new instance of \texttt{XrdSecCredentials} is created with the result of the serialization, to be handled over to the server.

Errors / failures

The client signals an error condition returning from \texttt{getCredentials()} with a null buffer. An error code is filled in the \texttt{XrdOucErrInfo} instance passed as argument to \texttt{getCredentials()}. The following error codes can be issued by the client. They are defined in \texttt{XrdSecgsi/XrdSecProtocolgsi.hh} and schematically described in Table 5.

Table 5. Errors issued by clients

<table>
<thead>
<tr>
<th>Error</th>
<th>Code</th>
<th>Situation</th>
</tr>
</thead>
<tbody>
<tr>
<td>kGSErrParseBuffer</td>
<td>10000</td>
<td>The received buffer could not be parsed</td>
</tr>
<tr>
<td>kGSErrDecodeBuffer</td>
<td>10001</td>
<td>Not enough memory for the global buffer</td>
</tr>
<tr>
<td>kGSErrBadProtocol</td>
<td>10003</td>
<td>Protocol ID does not match the expected one (gsi)</td>
</tr>
<tr>
<td>kGSErrCreateBucket</td>
<td>10004</td>
<td>Bucket can not be created; type in message string</td>
</tr>
<tr>
<td>kGSErrSerialBuffer</td>
<td>10007</td>
<td>Main buffer serialization fails</td>
</tr>
<tr>
<td>kGSErrBadRndmTag</td>
<td>10011</td>
<td>Random tag check failed</td>
</tr>
<tr>
<td>kGSErrNoCipher</td>
<td>10013</td>
<td>No cipher when expected</td>
</tr>
<tr>
<td>kGSErrBadOpt</td>
<td>10015</td>
<td>Unrecognized step</td>
</tr>
<tr>
<td>kGSErrNoBuffer</td>
<td>10019</td>
<td>No input parameters when expected</td>
</tr>
<tr>
<td>kGSErrNoPublic</td>
<td>10021</td>
<td>Problem extracting public component of cipher</td>
</tr>
<tr>
<td>kGSErrAddBucket</td>
<td>10022</td>
<td>Bucket can not be added; type in message string</td>
</tr>
<tr>
<td>kGSErrInit</td>
<td>10024</td>
<td>Error during protocol initialization</td>
</tr>
<tr>
<td>kGSErrError</td>
<td>10026</td>
<td>Generic error</td>
</tr>
</tbody>
</table>
Server side

Common pre-step processing

The information exchanged is first deserialized and then interpreted. The following steps are performed by the server on the buffer received by the client:

1. Update the TimeStamp in gsiHSVars;
2. Deserialize the received buffer (internal name bpar);
3. Check the protocol ID string to be “gsi”;
4. Determine the step required by the client

The remaining analysis of the received buckets depends on the client step.

Step processing

Step: kXGC_certreq

Received buffer

The information is contained in both the global and main buffers as described in Tables 7a and 7b.

Table 7a. Content of the global buffer bpar for step kXGC_certreq

<table>
<thead>
<tr>
<th>Bucket Type</th>
<th>Bucket content</th>
<th>Example</th>
<th>default</th>
<th>comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXRS_version</td>
<td>kXR_int32</td>
<td>10400</td>
<td></td>
<td>marshalled</td>
</tr>
<tr>
<td>kXRS_cryptomod</td>
<td>const char *</td>
<td>ssl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>kXRS_issuer_hash</td>
<td>const char *</td>
<td>5168735f.0</td>
<td>4339b4bc.0</td>
<td></td>
</tr>
<tr>
<td>kXRS_clnt_opts</td>
<td>kXR_int32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>kXRS_main</td>
<td>const char *</td>
<td>bmai (see Table 7b), plain text</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7b. Content of the global buffer bmai for step kXGC_certreq

<table>
<thead>
<tr>
<th>Bucket Type</th>
<th>Bucket content</th>
<th>Example, comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXRS_rtag</td>
<td>const char *</td>
<td>Challenge for the server</td>
</tr>
</tbody>
</table>
Actions performed

The server performs the following actions:

1. Extract from the global buffer the bucket of type kXRS_cryptomod with the list of cryptographic module names supported by the client; load the first supported crypto module available;
2. Extract from the global buffer the bucket of type kXRS_version with gsi protocol version run by the client; unmarshal the content; save the client gsi protocol version in the gsiHSvars instance;
3. Extract from the global buffer the bucket of type kXRS_issuer_hash with the hash of the issuer of the client certificate; load the related CA certificate;
4. Load the RSA private key of the server certificate in sessionKsig, to be used for signatures .
5. Extract from the global buffer the bucket of type kXRS_main with the main bucket; deserialize it;
6. Extract from the global buffer the bucket of type kXRS_clnt_opts with the client options; unmarshal the content; save the options in the gsiHSvars instance;

Preparation of the reply to kXGC_certreq

The following information is added the global buffer bpar:

4. A bucket of type kXRS_cipher with the server public part of the cipher;
5. A bucket of type kXRS_cipher_alg with the ‘|’ separated list of supported cipher algorithms, preferred first;
6. A bucket of type kXRS_md_alg with the ‘|’ separated list of supported message digest algorithms, preferred first;
7. A bucket of type kXRS_x509 with the server certificate;

The server sets nextstep for the server to be kXGS_cert. Return kgST_more .

Step: kXGC_cert

Received buffer

The information is contained in both the global and main buffers as described in Tables 8a and 8b.

Table 8a. Content of the global buffer bpar for step kXGC_cert

<table>
<thead>
<tr>
<th>Bucket Type</th>
<th>Bucket content</th>
<th>Example, comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXRS_cryptomod</td>
<td>const char *</td>
<td>ssl</td>
</tr>
<tr>
<td>kXRS_cipher_alg</td>
<td>const char *</td>
<td>aes-128-cbc#16</td>
</tr>
<tr>
<td>kXRS_md_alg</td>
<td>const char *</td>
<td>sha256</td>
</tr>
</tbody>
</table>
### Actions performed

The server performs the following actions:

1. Check the cached timestamp against the current timestamp; allow for 300 seconds skew;
2. Get from the global buffer the bucket of type `kXRS_cipher_alg` with the cipher algorithm chosen by the client; cross-check that it is supported locally
   a. Drop bucket `kXRS_cipher_alg` from the global buffer;
3. Get from the global buffer the bucket of type `kXRS_puk` with the client public key in PEM format and import it into `sessionKver`;
4. Get from the global buffer the bucket of type `kXRS_cipher` with the client public parameters for DH key agreement; decrypt the bucket with `sessionKver`; initialize the session cipher and store it in `sessionKey`;
   a. Drop bucket `kXRS_cipher` from the global buffer;
   b. Disable any delegation options if the DH public parameters are not signed;
5. Extract from the global buffer the bucket of type `kXRS_main` with the main bucket; decrypt the bucket with `sessionKey`; deserialize the main buffer;
6. Get from the global buffer the bucket of type `kXRS_x509` with the client proxy certificate and used it to finalize the client proxy certificate chain; verify the chain validity.
   a. Drop bucket `kXRS_x509` from the global buffer;
7. Extract the public key from the client certificate and make sure that it matches the one extracted from the dedicated bucket and previously saved in `sessionKver`;
8. Get from the global buffer the bucket of type `kXRS_md_alg` with the message digest algorithm chosen by the client; load it in `sessionMD`.
9. If a lookup of the `gridmap` file is required, check the `gridmap` file and fill `Entity.name` with the result; in case of failure, use the DN - or the DN hash, if required;

---

<table>
<thead>
<tr>
<th>Bucket Type</th>
<th>Bucket content</th>
<th>Example, comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>kXRS_cipher</code></td>
<td>Raw buffer</td>
<td>DH parameters, signed</td>
</tr>
<tr>
<td><code>kXRS_puk</code></td>
<td>const char *</td>
<td>Client public key, PEM format</td>
</tr>
<tr>
<td><code>kXRS_main</code></td>
<td>Raw buffer</td>
<td>Encrypted with <code>sessionKey</code></td>
</tr>
<tr>
<td><code>kXRS_signed_rtag</code></td>
<td>const char *</td>
<td>Server challenge signed with client private key</td>
</tr>
<tr>
<td><code>kXRS_rtag</code></td>
<td>const char *</td>
<td>Challenge for the server</td>
</tr>
<tr>
<td><code>kXRS_x509</code></td>
<td>const char *</td>
<td>Client proxy certificate, PEM format</td>
</tr>
</tbody>
</table>
10. If the extraction of the VOMS attributes is required, call the chosen function and fill the relevant fields in Entity with the result;
11. If authorization is required, run the relevant options.
12. If delegate proxies are requested
   a. Save the client proxy certificate chain;
   b. Prepare the proxy request (see dedicated section), save it into a bucket of type kXRS_x509_req and add it to the main buffer bmain.

Preparation of the reply to kXGC_cert
If delegate proxies are requested
   1. Set nextstep for the server to be kXGS_pxyreq.
   2. Return kgST_more;
Otherwise, set nextstep for the server to be kXGS_none; return kgST_ok or kgST_error.

Step: kXGC_sigpxy

Received buffer
The information is contained in both the global and main buffers as described in Tables 9a and 9b.

Table 9a. Content of the global buffer bpar for step kXGC_sigpxy

<table>
<thead>
<tr>
<th>Bucket Type</th>
<th>Bucket content</th>
<th>Example, comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXRS_cryptomod</td>
<td>const char *</td>
<td>ssl</td>
</tr>
<tr>
<td>kXRS_main</td>
<td>Raw buffer</td>
<td>Encrypted with sessionKey</td>
</tr>
</tbody>
</table>

Table 8b. Content of the global buffer bmai for step kXGC_cert

<table>
<thead>
<tr>
<th>Bucket Type</th>
<th>Bucket content</th>
<th>Example, comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXRS_x509</td>
<td>const char *</td>
<td>Client delegated proxy certificate, PEM format</td>
</tr>
<tr>
<td>kXRS_signed_rtag</td>
<td>const char *</td>
<td>Server challenge signed with client private key</td>
</tr>
</tbody>
</table>
Actions performed

The server performs the following actions:

1. Extract from the global buffer the bucket of type kXRS_main with the main bucket; decrypt the bucket with sessionKey; deserialize the main buffer;
2. Get from the global buffer the bucket of type kXRS_x509; this will contain either the client proxy private key or the full delegate proxy certificate (signed request); use to finalize the delegate client proxy certificate chain;
3. Honour the export options for the delegate proxies
   a. Export the delegated proxy as string and save it to Entity.creds;
   b. If a file is required, extract the bucket with the user name, type kXRS_user; prepare the file name, resolving the relevant place-holders, and save the delegated proxy to file.

Preparation of the reply to kXGC_sigpxy

Set nextstep for the server to be kXGS_none; return kgST_ok or kgST_error.

Common post-step processing

The following actions are performed after the processing of the step peculiarities:

8. The step nextstep is set both in the global buffer and in the main buffer;
9. If a random challenge was present in the received main buffer, in the form of a bucket of type kXRS_rtag, sign the challenge with the private key sessionKsig; the bucket type is updated to kXRS_signed_rtag;
10. A new random challenge is added to the main bucket as a bucket of type kXRS_rtag;
11. The new random challenge and the current time stamp are saved to a local cache;
12. The main buffer is serialized; the result of the serialization is used to update - or add - a bucket of type kXRS_main into the global buffer.
13. The main bucket is encrypted with session cipher sessionKey
   a. This does not apply to the first client step when sessionKey is not yet defined
14. The global buffer is serialized; a new instance of XrdSecCredentials is created with the result of the serialization, to be handled over to the client.

Errors / failures

Servers signals an error condition returning kgST_error from Authenticate(). An error code is filled in the XrdOucErrInfo instanced passed as argument to Authenticate(). The following error codes can be issued by the client. They are defined in XrdSecgsi/XrdSecProtocolgsi.hh and schematically described in Table 10.
Table 10. Errors issued by servers. An error message is also printed.

<table>
<thead>
<tr>
<th>Error</th>
<th>Code</th>
<th>Situation</th>
</tr>
</thead>
<tbody>
<tr>
<td>kGSErrParseBuffer</td>
<td>10000</td>
<td>The received buffer could not be parsed</td>
</tr>
<tr>
<td>kGSErrDecodeBuffer</td>
<td>10001</td>
<td>Not enough memory for the global buffer</td>
</tr>
<tr>
<td>kGSErrBadProtocol</td>
<td>10003</td>
<td>Protocol ID does not match the expected one (gsi)</td>
</tr>
<tr>
<td>kGSErrCreateBucket</td>
<td>10004</td>
<td>Bucket can not be created; type in message string</td>
</tr>
<tr>
<td>kGSErrSerialBuffer</td>
<td>10007</td>
<td>Main buffer serialization fails</td>
</tr>
<tr>
<td>kGSErrBadRndmTag</td>
<td>10011</td>
<td>Random tag check failed</td>
</tr>
<tr>
<td>kGSErrBadOpt</td>
<td>10015</td>
<td>Unrecognized step</td>
</tr>
<tr>
<td>kGSErrNoPublic</td>
<td>10021</td>
<td>Problem extracting public component of cipher</td>
</tr>
<tr>
<td>kGSErrAddBucket</td>
<td>10022</td>
<td>Bucket can not be added; type in message string</td>
</tr>
<tr>
<td>kGSErrInit</td>
<td>10024</td>
<td>Error during protocol initialization</td>
</tr>
<tr>
<td>kGSErrError</td>
<td>10026</td>
<td>Generic error; typically during sanity checks</td>
</tr>
</tbody>
</table>

(Delegated) Proxy certificates

Proxy certificates are X509 certificates of limited duration, signed by an end-entity certificate, and containing dedicated extensions [2]. A delegated proxy is a X509 proxy certificate issued by a X509 proxy certificate.

The creation of a proxy requires the following steps:
1. Load the end-entity certificate and private key
2. Create a X509 certificate request
3. Generate a private/public key pair; assign it to the X509 request
4. Generate a unique subject name for the proxy certificate:
   a. Duplicate the end-entity certificate subject name
   b. Generate a unique serial number
c. Add, to the duplicate certificate subject name, the unique serial number as new entry named “CN”
d. Set the generated subject name in the X509 request

5. Create the extension certProxyInfo
   a. Set the policy language on the extension to inheritALL [1]
      i. Policy language independent [1] and limited proxy not implemented
   b. Set the path length constraint, if required
   c. Set the extension OID to “1.3.6.1.5.5.7.1.14”
   d. Flag the extension as critical
   e. Format the extension data for addition to X509 request
   f. Create a stack of extensions; add the extension to the stack, add the stack to the X509 request
   g. Sign the X509 request with the public key of the X509 request

6. Build the proxy certificate
   a. Create an empty X509 certificate
   b. Set the version number to 3 (meaning: ‘extension are present’)
   c. Set serial number, subject name and key from the X509 request
   d. Set the issuer name to the the subject name of the end-entity certificate
   e. Adjust the validity according to needs
   f. Transfer all the extensions from the end-entity certificate
   g. Add the certProxyInfo extension from the X509 request
   h. Sign the proxy with the end-entity certificate key

The process can be repeated starting from a X509 proxy certificate instead of a X509 end-entity certificate; that is what is done to generated a delegate proxy.

Server identity verification

A crucial part to avoid man-in-the-middle attacks is the client verification of server identity. The basic idea is that the client knows the name of the server it is contacting and expects to find this name in the DN of the server certificate. Complications arise when hostname aliases are used, and/or when the same server certificate is used by more servers, making use of the Subject Alternative Name (SAN) support.

Support for SAN matching is introduced in v4.9, together with alternative ways to resolve the hostname on the client, without necessarily relying on the DNS. Despite the version, the client has the possibility to defined exceptions via the environment variable XrdSecGSISRVNAMES, a comma-separated list of allowed/disallowed names, supporting wild-cards.
References

Appendix A - Relevant parts of header files

XrdSecEntity.hh

#define XrdSecPROTOIDSIZE 8

class XrdSecEntity
{
public:
    char prot[XrdSecPROTOIDSIZE]; // Protocol used
    char *name;                   // Entity's name
    char *host;                   // Entity's host name dnr dependent
    char *vorg;                   // Entity's virtual organization
    char *role;                   // Entity's role
    char *grps;                   // Entity's group names
    char *endorsements;           // Protocol specific endorsements
    char *moninfo;                // Additional information for monitoring
    char *creds;                  // Raw client credentials or certificate
    int credslen;                 // Length of the 'creds' field
    int rsvd;                     // Reserved field
    XrdNetAddrInfo *addrInfo;     // Connection details from getProtocol
    const char *tident;           // Trace identifier always preset
    void *sessvar;                // Plugin settable storage pointer
                                  // that is common to the session. Free
                                  // it in your XrdSfsFileSystem::Disc()
                                  // implementation, as needed.

    XrdSecEntity(const char *pName = "")
    {Reset();
        strncpy(prot, pName, XrdSecPROTOIDSIZE-1);
        prot[XrdSecPROTOIDSIZE-1] = '\0';
        addrInfo = 0; name = 0; host = 0; vorg = 0;
        role = 0; grps = 0; endorsements = 0;
        moninfo = 0; creds = 0; credslen = 0;
        rsvd = 0; addrInfo = 0; tident = 0; sessvar = 0;
    }

    ~XrdSecEntity() {}
    void Reset() {
        memset( prot, 0, XrdSecPROTOIDSIZE );
        name = 0; host = 0; vorg = 0;
        role = 0; grps = 0; endorsements = 0;
        moninfo = 0; creds = 0; credslen = 0;
        rsvd = 0; addrInfo = 0; tident = 0; sessvar = 0;
    }
};
struct XrdSecBuffer
{
    int size;   //!< Size of the buffer or length of data in the buffer
    char *buffer;  //!< Pointer to the buffer
    XrdSecBuffer(char *bp=0, int sz=0) : size(sz), buffer(bp), membuf(bp) {}
    ~XrdSecBuffer() {if (membuf) free(membuf);}  
private:
    char *membuf; // Stable copy of the buffer address
};

typedef XrdSecBuffer XrdSecCredentials;
typedef XrdSecBuffer XrdSecParameters;

class XrdSecProtocol
{
public:

    //******************************************************************************
    //! Structure holding the entity's identification. It is filled in by a
    //!   successful call to Authenticate() (i.e. it returns 0).
    //******************************************************************************
    XrdSecEntity Entity;

    //******************************************************************************
    //! Authenticate a client.
    //!******************************************************************************
    virtual int Authenticate (XrdSecCredentials *cred,
                               XrdSecParameters **parms,
                               XrdOucErrInfo *einfo=0)=0;

};

XRootD GSI Protocol Specifications  27
virtual XrdSecCredentials *getCredentials(XrdSecParameters *parm=0, XrdOucErrInfo *einfo=0)=0;

virtual int Encrypt(const char *inbuff, int inlen, XrdSecBuffer **outbuff) {
    (void) inbuff; (void) inlen; (void) outbuff;
    return -ENOTSUP;
}

virtual int Decrypt(const char *inbuff, int inlen, XrdSecBuffer **outbuff) {
    (void) inbuff; (void) inlen; (void) outbuff;
    return -ENOTSUP;
}
virtual int Decrypt(const char *inbuff, // Data to be decrypted
    int inlen, // Length of data in inbuff
    XrdSecBuffer **outbuff // Buffer for decrypted data
)
{
    (void) inbuff; (void) inlen; (void) outbuff;
    return -ENOTSUP;
}

#ifndef XRRXSEC_H
#endif

virtual int Sign(const char *inbuff, // Data to be signed
                 int inlen, // Length of data in inbuff
    XrdSecBuffer **outbuff // Buffer for the signature
)
{
    (void) inbuff; (void) inlen; (void) outbuff;
    return -ENOTSUP;
}

virtual int Verify(const char *inbuff, // Data to be decrypted
        int inlen, // Length of data in inbuff
        const char *sigbuff, // Buffer for signature
        int siglen) // Length if signature
{
    (void) inbuff; (void) inlen; (void) sigbuff; (void) siglen;
return -ENOTSUP;
}

// ! Get the current encryption key (i.e. session key)
//@
//@ @param buff    buffer to hold the key, and may be null.
//@ @param size    size of the buffer.
//@
//@ @returns <  0 Failed, returned value if -errno (see Encrypt)
//@      >= 0 The size of the encryption key. The supplied buffer of length
//@            size hold the key. If the buffer address is supplied, the
//@            key is placed in the buffer.
//@
//@ //-------------------------------------------------------------------------------
virtual int     getKey(char *buff = 0, int size = 0)
{
   (void) buff; (void) size;
   return -ENOTSUP;
}

// ! Set the current encryption key
//@
//@ @param buff    buffer that holds the key.
//@ @param size    size of the key.
//@
//@ @returns: < 0 Failed, returned value if -errno (see Encrypt)
//@        = 0 The new key has been set.
//@ //-------------------------------------------------------------------------------
virtual int     setKey(char *buff, int size)
{
   (void) buff; (void) size;
   return -ENOTSUP;
}

// ! Delete the protocol object. DO NOT use C++ delete() on this object.
//@ //-------------------------------------------------------------------------------
virtual void    Delete()=0; // Normally does "delete this"

//@ ! Constructor
//@ //-------------------------------------------------------------------------------
XrdSecProtocol(const char *pName) : Entity(pName) {}
protected:
//----------------------------------------------------------------------------------------
//! Destructor (prevents use of direct delete).
//----------------------------------------------------------------------------------------
virtual ~XrdSecProtocol() {} 
};

XrdSutBucket.hh

class XrdSutBucket
{
public:
  kXR_int32 type;
  kXR_int32 size;
  char *buffer;

  XrdSutBucket(char *bp=0, int sz=0, int ty=0);
  XrdSutBucket(XrdOucString &s, int ty=0);
  XrdSutBucket(XrdSutBucket &b);
  virtual ~XrdSutBucket() {if (membuf) delete[] membuf;}

  void Update(char *nb = 0, int ns = 0, int ty = 0); // Uses 'nb'
  int Update(XrdOucString &s, int ty = 0);
  int SetBuf(const char *nb = 0, int ns = 0);         // Duplicates 'nb'

  void Dump(int opt = 1);
  void ToString(XrdOucString &s);

  // Equality operator
  int operator==(const XrdSutBucket &b);

  // Inequality operator
  int operator!=(const XrdSutBucket &b) { return !(this == b); }

private:
  char *membuf;
};
class XrdSutBuckListNode {
private:
    XrdSutBucket *buck;
    XrdSutBuckListNode *next;
public:
    XrdSutBuckListNode(XrdSutBucket *b = 0, XrdSutBuckListNode *n = 0) {
        buck = b; next = n;
    }
    virtual ~XrdSutBuckListNode() { }
    XrdSutBucket *Buck() const { return buck; }
    XrdSutBuckListNode *Next() const { return next; }
    void SetNext(XrdSutBuckListNode *n) { next = n; }
};

class XrdSutBuckList {
private:
    XrdSutBuckListNode *begin;
    XrdSutBuckListNode *current;
    XrdSutBuckListNode *end;
    XrdSutBuckListNode *previous;
    int size;
    XrdSutBuckListNode *Find(XrdSutBucket *b);
public:
    XrdSutBuckList(XrdSutBucket *b = 0);
    virtual ~XrdSutBuckList();
    int Size() const { return size; }
    XrdSutBucket *End() const { return end->Buck(); }
    void PutInFront(XrdSutBucket *b);
    void PushBack(XrdSutBucket *b);
    void Remove(XrdSutBucket *b);
    XrdSutBucket *Begin();
    XrdSutBucket *Next();
};
class XrdSutBuffer {
private:
    XrdSutBuckList  fBuckets;
    XrdOucString   fOptions;
    XrdOucString   fProtocol;
    kXR_int32      fStep;

public:
    XrdSutBuffer(const char *prot, const char *opts = 0)
        { fOptions = opts; fProtocol = prot; fStep = 0; }
    XrdSutBuffer(const char *buffer, kXR_int32 length);
    virtual ~XrdSutBuffer();

    int         AddBucket(char *bp=0, int sz=0, int ty=0)
    { XrdSutBucket *b = new XrdSutBucket(bp,sz,ty);
      if (b) { fBuckets.PushBack(b); return 0; } return -1; }
    int         AddBucket(XrdOucString s, int ty=0)
    { XrdSutBucket *b = new XrdSutBucket(s,ty);
      if (b) { fBuckets.PushBack(b); return 0; } return -1; }
    int         AddBucket(XrdSutBucket *b)
    { if (b) { fBuckets.PushBack(b); return 0; } return -1; }

    int         UpdateBucket(const char *bp, int sz, int ty);
    int         UpdateBucket(XrdOucString s, int ty);
    int         UpdateBucket(XrdSutBucket *b)
    { if (b) { fBuckets.PushBack(b); return 0; } return -1; }

    // Remove from the list, to avoid destroy by ~XrdSutBuffer
    void        Remove(XrdSutBucket *b) { fBuckets.Remove(b); } 

    void        Dump(const char *stepstr = 0);
    void        Message(const char *prepose = 0);
    int         Serialized(char **buffer, char opt = 'n');
    void        Deactivate(kXR_int32 type);  // Deactivate bucket (type=-1 for cleanup)

    // To fill / access buckets containing 4-byte integers (status codes, versions ...)
    kXR_int32   MarshalBucket(kXR_int32 type, kXR_int32 code);
    kXR_int32   UnmarshalBucket(kXR_int32 type, kXR_int32 &code);

    XrdSutBucket *GetBucket(kXR_int32 type, const char *tag = 0);
    XrdSutBuckList *GetBuckList() const { return (XrdSutBuckList *)&fBuckets; }

    int         GetNBuckets() const     { return fBuckets.Size(); }
    const char *GetOptions() const     { return fOptions.c_str(); }
    const char *GetProtocol() const    { return fProtocol.c_str(); }
    int         GetStep() const         { return (int)fStep; }
    void        SetStep(int s)   { fStep = (kXR_int32)s; }
    void        IncrementStep()  { fStep++; }
};
XrdSutAux.hh (excerpt)

```
#define XrdSutMAXBUF  4096
#define XrdSutMAXPPT   512
#define XrdSutMAXBUCKS   10
#define XrdSutMAXINT64LEN  25
#define XrdSutPRINTLEN  100

enum kXRSBucketTypes {
    kXRS_none   = 0,      // end-of-vector
    kXRS_inactive = 1,     // inactive (dropped at serialization)
    kXRS_cryptomod = 3000, // 3000 Name of crypto module to use
    kXRS_main,      // 3001 Main buffer
    kXRS_srv_seal,  // 3002 Server secrets sent back as they are
    kXRS_clnt_seal, // 3003 Client secrets sent back as they are
    kXRS_puk,       // 3004 Public Key
    kXRS_cipher,    // 3005 Cipher
    kXRS_rtag,      // 3006 Random Tag
    kXRS_signed_rtag, // 3007 Random Tag signed by the client
    kXRS_user,      // 3008 User name
    kXRS_host,      // 3009 Remote Host name
    kXRS_creds,     // 3010 Credentials (password, ...)
    kXRS_message,   // 3011 Message (null-terminated string)
    kXRS_srvID,     // 3012 Server unique ID
    kXRS_sessionID, // 3013 Handshake session ID
    kXRS_version,   // 3014 Package version
    kXRS_status,    // 3015 Status code
    kXRS_localstatus, // 3016 Status code(s) saved in sealed buffer
    kXRS_othercreds, // 3017 Alternative creds (e.g. other crypto)
    kXRS_cache_idx, // 3018 Cache entry index
    kXRS_clnt_opts, // 3019 Client options, if any
    kXRS_error_code, // 3020 Error code
    kXRS_timestamp, // 3021 Time stamp
    kXRS_x509, // 3022 X509 certificate
    kXRSIssuerHash, // 3023 Issuer hash
    kXRS_x509req, // 3024 X509 certificate request
    kXRS_cipheralg, // 3025 Cipher algorithm (list)
    kXRS_mdalg, // 3026 MD algorithm (list)
    kXRS_afsinfo, // 3027 AFS information
    kXRSReserved // 3027 Reserved
};
```
XrdSecProtoColgsi.hh

/******************************************************************************/
/*                               D e f i n e s                                */
/******************************************************************************/
typedef XrdOucString String;
typedef XrdCryptogsiX509Chain X509Chain;

#define XrdSecPROTOIDENT "gsi"
#define XrdSecPROTOIDLEN sizeof(XrdSecPROTOIDENT)
#define XrdSecgsiVERSION 10300
#define XrdSecNOIPCHK 0x0001
#define XrdSecDEBUG 0x1000
#define XrdCryptoMax 10

#define kMAXBUFLEN 1024

// Message codes either returned by server or included in buffers
enum kgsiStatus {
    kgST_error  = -1,  // error occured
    kgST_ok     =  0,  // ok
    kgST_more   =  1,  // need more info
};

// Client steps
enum kgsiClientSteps {
    kXGC_none = 0,
    kXGC_certreq  = 1000,  // 1000: request server certificate
    kXGC_cert,  // 1001: packet with (proxy) certificate
    kXGC_sigpxy,  // 1002: packet with signed proxy certificate
    kXGC_reserved  
};

// Server steps
enum kgsiServerSteps {
    kXGS_none = 0,
    kXGS_init   = 2000,  // 2000: fake code used the first time
    kXGS_cert,  // 2001: packet with certificate
    kXGS_pxyreq,  // 2002: packet with proxy req to be signed
    kXGS_reserved
};

// Handshake options
enum kgsiHandshakeOpts {
    kOptsDlgPxy = 1,  // 0x0001: Ask for a delegated proxy
    kOptsFwdPxy = 2,  // 0x0002: Forward local proxy
}
kOptsSigReq = 4,     // 0x0004: Accept to sign delegated proxy
kOptsSrvReq = 8,     // 0x0008: Server request for delegated proxy
kOptsPxFile = 16,     // 0x0010: Save delegated proxies in file
kOptsDelChn = 32,     // 0x0020: Delete chain
kOptsPxCred = 64      // 0x0040: Save delegated proxies as credentials
};

// Error codes
eenum kgsiErrors {
    kGSErrParseBuffer = 10000,       // 10000
    kGSErrDecodeBuffer,              // 10001
    kGSErrLoadCrypto,                // 10002
    kGSErrBadProtocol,               // 10003
    kGSErrCreateBucket,              // 10004
    kGSErrDuplicateBucket,           // 10005
    kGSErrCreateBuffer,              // 10006
    kGSErrSerialBuffer,              // 10007
    kGSErrGenCipher,                 // 10008
    kGSErrExportPuK,                 // 10009
    kGSErrEncRndmTag,                // 10010
    kGSErrBadRndmTag,                // 10011
    kGSErrNoRndmTag,                 // 10012
    kGSErrNoCipher,                  // 10013
    kGSErrNoCreds,                   // 10014
    kGSErrBadOpt,                    // 10015
    kGSErrMarshal,                   // 10016
    kGSErrUnmarshal,                 // 10017
    kGSErrSaveCreds,                 // 10018
    kGSErrNoBuffer,                  // 10019
    kGSErrRefCipher,                 // 10020
    kGSErrNoPublic,                  // 10021
    kGSErrAddBucket,                 // 10022
    kGSErrFinCipher,                 // 10023
    kGSErrInit,                      // 10024
    kGSErrBadCreds,                  // 10025
    kGSErrError                      // 10026
};

#define REL1(x)     { if (x) delete x; }
#define REL2(x,y)   { if (x) delete x; if (y) delete y; }
#define REL3(x,y,z) { if (x) delete x; if (y) delete y; if (z) delete z; }

#define SafeDelete(x) { if (x) delete x ; x = 0; }
#define SafeDelArray(x) { if (x) delete [] x ; x = 0; }
#define SafeFree(x) { if (x) free(x) ; x = 0; }

// External functions for generic mapping
typedef char *(*XrdSecgsiGMAP_t)(const char *, int);
typedef int (*XrdSecgsiAuthz_t)(XrdSecEntity &);
typedef int (*XrdSecgsiAuthzInit_t)(const char *);
typedef int (*XrdSecgsiAuthzKey_t)(XrdSecEntity &, char **);

// VOMS extraction
typedef XrdSecgsiAuthz_t XrdSecgsiVOMS_t;
typedef XrdSecgsiAuthzInit_t XrdSecgsiVOMSInit_t;

// This a small class to set the relevant options in one go
class XrdOucGMap;
class XrdOucTrace;
class gsiOptions {

public:
  short debug; // [cs] debug flag
  char mode;  // [cs] 'c' or 's'
  char *clist; // [s] list of crypto modules ["ssl"]
  char *certdir; // [cs] dir with CA info [/etc/grid-security/certificates]
  char *crldir; // [cs] dir with CRL info [/etc/grid-security/certificates]
  char *crlext; // [cs] extension of CRL files [.r0]
  char *cert;  // [s] server certificate [/etc/grid-security/root/rootcert.pem]
    // [c] user certificate [$HOME/.globus/usercert.pem]
  char *key;   // [s] server private key [/etc/grid-security/root/rootkey.pem]
    // [c] user private key [$HOME/.globus/userkey.pem]
  char *cipher; // [s] list of ciphers [aes-128-cbc:bf-cbc:des-ede3-cbc]
  char *md;    // [s] list of MDs [sha256:md5]
  int crl;    // [cs] check level of CRL's [1]
  int crlrefresh; // [cs] CRL refresh or expiration period in secs [1 day]
  char *proxy; // [c] user proxy [/tmp/x509up_u<uid>]
  char *valid; // [c] proxy validity [12:00]
  int deplen; // [c] depth of signature path for proxies [0]
  int bits;  // [c] bits in PKI for proxies [512]
  char *gridmap; // [s] gridmap file [/etc/grid-security/gridmap]
  int gmapto; // [s] validity in secs of grid-map cache entries [600 s]
  char *gmapfun; // [s] file with the function to map DN to usernames [0]
  char *gmapfunparms; // [s] parameters for the function to map DN to usernames [0]
  char *authzfun; // [s] file with the function to fill entities [0]
  char *authzfunparms; // [s] parameters for the function to fill entities [0]
  int authzto; // [s] validity in secs of authz cache entries [-1 => unlimited]
  int ogmap;  // [s] gridmap file checking option
  int dlgpxy; // [c] explicitely ask the creation of a delegated proxy; default 0
    // [s] ask client for proxies; default: do not accept delegated proxies
  int sigpxy; // [c] accept delegated proxy requests
  char *srvnames; // [c] '!' separated list of allowed server names
  char *exppxy; // [s] template for the exported file with proxies
  int authzpxy; // [s] if 1 make proxy available in exported form in the
    //     field of the XrdSecEntity object for use in XrdAcc
  int vomsat; // [s] 0 do not look for; 1 extract if any
  char *vomsfun; // [s] file with the function to fill VOMS [0]
  char *vomsfunparms; // [s] parameters for the function to fill VOMS [0]
int moninfo; // [s] 0 do not look for; 1 use DN as default
int hashcomp; // [cs] 1 send hash names with both algorithms;
0 send only the default [1]

bool trustdns; // [cs] 'true' if DNS is trusted [true]

gsiOptions() { debug = -1; mode = 's'; clist = 0;
certdir = 0; crldir = 0; crlext = 0; cert = 0; key = 0;
cipher = 0; md = 0; ca = 1 ; crl = 1; crlrefresh = 86400;
proxy = 0; valid = 0; deplen = 0; bits = 512;
gridmap = 0; gmapto = 600;
gmapfun = 0; gmapfunparms = 0; authzfun = 0;
authzfunparms = 0; authzto = -1;
ogmap = 1; dlgpxy = 0; sigpxy = 1; srvnames = 0;
exppxy = 0; authzpxy = 0;
vomap = 1; vomsfun = 0; vomsfunparms = 0; moninfo = 0; hashcomp =
1; trustdns = true; }
virtual ~gsiOptions() { } // Cleanup inside XrdSecProtocolgsiInit
void Print(XrdOucTrace *t); // Print summary of gsi option status
};

class XrdSecProtocolgsi;
class gsiHSVars;

// From a proxy query
typedef struct {
  X509Chain *chain;
  XrdCryptoRSA *ksig;
  XrdSutBucket *cbck;
} ProxyOut_t;

// To query proxies
typedef struct {
  const char *cert;
  const char *key;
  const char *certdir;
  const char *out;
  const char *valid;
  int deplen;
  int bits;
} ProxyIn_t;

template<class T>
class GSIStack {
public:
  void Add(T *t) {
    char k[40]; snprintf(k, 40, "%p", t);
    mtx.Lock();
    if (!stack.Find(k)) stack.Add(k, t, 0, Hash_count); // We need an additional
    count
  }
};

XRootD GSI Protocol Specifications

38
class XrdSecProtocolgsi : public XrdSecProtocol
{
friend class gsiOptions;
friend class gsiHSVars;
public:
    int Authenticate (XrdSecCredentials *cred,
                        XrdSecParameters **parms,
                        XrdOucErrInfo     *einfo=0);
    XrdSecCredentials *getCredentials(XrdSecParameters *parm=0,
                                        XrdOucErrInfo     *einfo=0);
    XrdSecProtocolgsi(int opts, const char *hname, XrdNetAddrInfo &endPoint,
                       const char *parms = 0);
    virtual ~XrdSecProtocolgsi() {} // Delete() does it all

    // Initialization methods
    static char      *Init(gsiOptions o, XrdOucErrInfo *erp);

    void              Delete();

    // Encrypt / Decrypt methods
    int                Encrypt(const char *inbuf, int inlen,
                               XrdSecBuffer **outbuf);
    int                Decrypt(const char *inbuf, int inlen,
                               XrdSecBuffer **outbuf);

    // Sign / Verify methods
    int                Sign(const char *inbuf, int inlen,
                            XrdSecBuffer **outbuf);
    int                Verify(const char *inbuf, int inlen,
                                const char *sigbuf, int siglen);

XRootD GSI Protocol Specifications
// Export session key
int getKey(char *kbuf=0, int klen=0);

// Import a key
int setKey(char *kbuf, int klen);

// Enable tracing
static XrdOucTrace *EnableTracing();

private:
    XrdNetAddrInfo epAddr;

    // Static members initialized at startup
    static XrdSysMutex gsiContext;
    static String CAdir;
    static String CRLdir;
    static String DefCRLext;
    static String SrvCert;
    static String SrvKey;
    static String UsrProxy;
    static String UsrCert;
    static String UsrKey;
    static String PxyValid;
    static int DepLength;
    static int DefBits;
    static int CACheck;
    static int CRLCheck;
    static int CRLDownload;
    static int CRLRefresh;
    static String DefCrypto;
    static String DefCipher;
    static String DefMD;
    static String DefError;
    static String GMAPFile;
    static int GMAPOpt;
    static bool GMAPuseDNname;
    static int GMAPCacheTimeOut;
    static XrdSecgsiGMAP_t GMAPFun;
    static XrdSecgsiAuthz_t AuthzFun;
    static XrdSecgsiAuthzKey_t AuthzKey;
    static int AuthzCertFmt;
    static int AuthzCacheTimeOut;
    static int PxyReqOpts;
    static int AuthzPxyWhat;
    static int AuthzPxyWhere;
    static String SrvAllowedNames;
    static int VOMSAttrOpt;
    static XrdSecgsiVOMS_t VOMSFun;
    static int VOMSCertFmt;
    static int MonInfoOpt;
    static bool HashCompatibility;
static bool TrustDNS;

// Crypto related info
static int ncrypt;                // Number of factories
static XrdCryptoFactory *cryptF[XrdCryptoMax];  // their hooks
static int cryptID[XrdCryptoMax];    // their IDs
static String cryptName[XrdCryptoMax]; // their names
static XrdCryptoCipher *refcip[XrdCryptoMax]; // ref for session ciphers

// Caches
static XrdSutCache cacheCA;   // Info about trusted CA's
static XrdSutCache cacheCert; // Server certificates info cache
static XrdSutCache cachePxy;  // Client proxies cache;
static XrdSutCache cacheGMAPFun; // Cache for entries mapped by GMAPFun
static XrdSutCache cacheAuthzFun; // Cache for entities filled by AuthzFun

// Services
static XrdOucGMap *servGMap; // Grid mapping service

// CA and CRL stacks
static GSIStack<XrdCryptoX509Chain> stackCA; // Stack of CA in use
static GSIStack<XrdCryptoX509Crl> stackCRL; // Stack of CRL in use

// GMAP control vars
static time_t lastGMAPCheck; // time of last check on GMAP
static XrdSysMutex mutexGMAP; // mutex to control GMAP reloads

// Running options / settings
static int Debug;          // [CS] Debug level
static bool Server;        // [CS] If server mode
static int TimeSkew;       // [CS] Allowed skew in secs for time stamps

// for error logging and tracing
static XrdSysLogger Logger;
static XrdSysError eDest;
static XrdOucTrace *GSITrace;

// Information local to this instance
int options;
XrdCryptoFactory *sessionCF;  // Chosen crypto factory
XrdCryptoCipher *sessionKey;  // Session Key (result of the handshake)
XrdSutBucket *bucketKey;      // Bucket with the key in export form
XrdCryptoMsgDigest *sessionMD; // Message Digest instance
XrdCryptoRSA *sessionKsig;    // RSA key to sign
XrdCryptoRSA *sessionKver;    // RSA key to verify
X509Chain *proxyChain;        // Chain with the delegated proxy on servers
bool srvMode;                 // TRUE if server mode
// Temporary Handshake local info
gsiHSVars *hs;

// Parsing received buffers: client
int ParseClientInput(XrdSutBuffer *br, XrdSutBuffer **bm,
                     String &emsg);
int ClientDoInit(XrdSutBuffer *br, XrdSutBuffer **bm,
                  String &cmsg);
int ClientDoCert(XrdSutBuffer *br, XrdSutBuffer **bm,
                 String &cmsg);
int ClientDoPxyreq(XrdSutBuffer *br, XrdSutBuffer **bm,
                   String &cmsg);

// Parsing received buffers: server
int ParseServerInput(XrdSutBuffer *br, XrdSutBuffer **bm,
                      String &cmsg);
int ServerDoCertreq(XrdSutBuffer *br, XrdSutBuffer **bm,
                    String &cmsg);
int ServerDoCert(XrdSutBuffer *br, XrdSutBuffer **bm,
                 String &cmsg);
int ServerDoSigpxy(XrdSutBuffer *br, XrdSutBuffer **bm,
                   String &cmsg);

// Auxilliary functions
int ParseCrypto(String cryptlist);
int ParseCAlist(String calist);

// Load CA certificates
static int GetCA(const char *cahash,
                 XrdCryptoFactory *cryptof, gsiHSVars *hs = 0);
static String GetCApath(const char *cahash);
static bool VerifyCA(int opt, X509Chain *cca, XrdCryptoFactory *cf);
static int VerifyCRL(XrdCryptoX509Crl *crl1,
                     XrdCryptoX509 *xca, XrdOucString crldir,
                     XrdCryptoFactory *CF, int hashalg);
bool ServerCertNameOK(const char *subject, String &e);
static XrdSutCacheEntry *GetSrvCertEnt(XrdSutCERef &gcref,
                                       XrdCryptoFactory *cf,
                                       time_t timestamp, String &cal);

// Load CRLs
static XrdCryptoX509Crl *LoadCRL(XrdCryptoX509 *xca, const char *sjhash,
                                  XrdCryptoFactory *CF, int dwld, int &err);

// Updating proxies
static int QueryProxy(bool checkcache, XrdSutCache *cache, const char *tag,
                       XrdCryptoFactory *cf, time_t timestamp,
                       ProxyIn_t *pi, ProxyOut_t *po);
static int InitProxy(ProxyIn_t *pi, XrdCryptoFactory *cf,
                    X509Chain *ch = 0, XrdCryptoRSA **key = 0);
// Error functions
static void    ErrF(XrdOucErrInfo *einfo, kXR_int32 ecode,
    const char *msg1, const char *msg2 = 0,
    const char *msg3 = 0);
XrdSecCredentials *ErrC(XrdOucErrInfo *einfo, XrdSutBuffer *b1,
    XrdSutBuffer *b2,XrdSutBuffer *b3,
    kXR_int32 ecode, const char *msg1 = 0,
    const char *msg2 = 0, const char *msg3 = 0);
int            ErrS(String ID, XrdOucErrInfo *einfo, XrdSutBuffer *b1,
    XrdSutBuffer *b2, XrdSutBuffer *b3,
    kXR_int32 ecode, const char *msg1 = 0,
    const char *msg2 = 0, const char *msg3 = 0);

// Check Time stamp
bool           CheckTimeStamp(XrdSutBuffer *b, int skew, String &emsg);

// Check random challenge
bool           CheckRtag(XrdSutBuffer *bm, String &emsg);

// Auxilliary methods
int            AddSerialized(char opt, kXR_int32 step, String ID,
    XrdSutBuffer *b1s, XrdSutBuffer *buf,
    kXR_int32 type, XrdCryptoCipher *cip);

// Grid map cache handling
static XrdSecgsiGMAP_t            // Load alternative function for mapping
    LoadGMAPFun(const char *plugin, const char *parms);
static XrdSecgsiAuthz_t           // Load alternative function to fill XrdSecEntity
    LoadAuthzFun(const char *plugin, const char *parms, int &fmt);
static XrdSecgsiVOMS_t           // Load alternative function to extract VOMS
    LoadVOMSFun(const char *plugin, const char *parms, int &fmt);
static void    //Lookup info for DN
    QueryGMAP(XrdCryptoX509Chain* chain, int now, String &name);

// Entity handling
void CopyEntity(XrdSecEntity *in, XrdSecEntity *out, int *lout = 0);
void FreeEntity(XrdSecEntity *in);

// VOMS parsing
int ExtractVOMS(X509Chain *c, XrdSecEntity &ent);
};

class gsiHSVars {
public:
    int               Iter;          // iteration number
    time_t            TimeStamp;     // Time of last call
    String            CryptoMod;     // crypto module in use
    int               RemVers;       // Version run by remote counterpart
    XrdCryptoCipher  *Rcip;          // reference cipher
    XrdSutBucket     *Cbck;          // Bucket with the certificate in export form
};
String            ID;            // Handshake ID (dummy for clients)
XrdSutPFEntry    *Cref;          // Cache reference
XrdSutPFEntry    *Pent;          // Pointer to relevant file entry
X509Chain        *Chain;         // Chain to be eventually verified
XrdCryptoX509Crl *Crl;           // Pointer to CRL, if required
X509Chain        *PxyChain;      // Proxy Chain on clients
bool              RtagOK;        // Rndm tag checked / not checked
bool              Tty;           // Terminal attached / not attached
int               LastStep;      // Step required at previous iteration
int               Options;       // Handshake options;
int               HashAlg;       // Hash algorithm of peer hash name;
XrdSutBuffer     *Parms;         // Buffer with server parms on first iteration

gsiHSVars() { Iter = 0; TimeStamp = -1; CryptoMod = "";
RemVers = -1; Rcip = 0;
Cbck = 0;
ID = ""; Cref = 0; Pent = 0; Chain = 0; Crl = 0; PxyChain = 0;
RtagOK = 0; Tty = 0; LastStep = 0; Options = 0; HashAlg = 0;
Parms = 0;}

~gsiHSVars() { SafeDelete(Cref);
if (Options & kOptsDelChn) {
    // Do not delete the CA certificate in the cached reference
    if (Chain) Chain->Cleanup(1);
    SafeDelete(Chain);
}
if (Crl) {
    // This decreases the counter and actually deletes the object
    // only when no instance is using it
    XrdSecProtocolgsi::stackCRL.Del(Crl);
    Crl = 0;
}
// The proxy chain is owned by the proxy cache; invalid proxies are
// detected (and eventually removed) by QueryProxy
PxyChain = 0;
SafeDelete(Parms); }
void Dump(XrdSecProtocolgsi *p = 0);
Appendix B - Details of bucket types

Table 1. Bucket types as defined in `XrdSutAux.hh`. Last column indicates those used by `gsi`.

<table>
<thead>
<tr>
<th>Name</th>
<th>Number</th>
<th>Description</th>
<th>Used by gsi</th>
</tr>
</thead>
<tbody>
<tr>
<td>kXRS_none</td>
<td>0</td>
<td>end-of-vector</td>
<td></td>
</tr>
<tr>
<td>kXRS_inactive</td>
<td>1</td>
<td>inactive (dropped at serialization)</td>
<td>y</td>
</tr>
<tr>
<td>kXRS_cryptomod</td>
<td>3000</td>
<td>Name of crypto module to use</td>
<td>y</td>
</tr>
<tr>
<td>kXRS_main</td>
<td>3001</td>
<td>Main buffer</td>
<td>y</td>
</tr>
<tr>
<td>kXRS_srv_seal</td>
<td>3002</td>
<td>Server secrets sent back as they are</td>
<td>y</td>
</tr>
<tr>
<td>kXRS_clnt_seal</td>
<td>3003</td>
<td>Client secrets sent back as they are</td>
<td>y</td>
</tr>
<tr>
<td>kXRS_puk</td>
<td>3004</td>
<td>Public Key</td>
<td>y</td>
</tr>
<tr>
<td>kXRS_cipher</td>
<td>3005</td>
<td>Cipher</td>
<td>y</td>
</tr>
<tr>
<td>kXRS_rtag</td>
<td>3006</td>
<td>Random Tag</td>
<td>y</td>
</tr>
<tr>
<td>kXRS_signed_rtag</td>
<td>3007</td>
<td>Random Tag signed by the client</td>
<td>y</td>
</tr>
<tr>
<td>kXRS_user</td>
<td>3008</td>
<td>User name</td>
<td></td>
</tr>
<tr>
<td>kXRS_host</td>
<td>3009</td>
<td>Remote Host name</td>
<td></td>
</tr>
<tr>
<td>kXRS_creds</td>
<td>3010</td>
<td>Credentials (password, ...)</td>
<td></td>
</tr>
<tr>
<td>kXRS_message</td>
<td>3011</td>
<td>Message (null-terminated string)</td>
<td></td>
</tr>
<tr>
<td>kXRS_srvID</td>
<td>3012</td>
<td>Server unique ID</td>
<td></td>
</tr>
<tr>
<td>kXRS_sessionID</td>
<td>3013</td>
<td>Handshake session ID</td>
<td></td>
</tr>
<tr>
<td>kXRS_version</td>
<td>3014</td>
<td>Package version</td>
<td>y</td>
</tr>
<tr>
<td>kXRS_status</td>
<td>3015</td>
<td>Status code</td>
<td>y</td>
</tr>
<tr>
<td>kXRS_localstatus</td>
<td>3016</td>
<td>Status code(s) saved in sealed buffer</td>
<td></td>
</tr>
<tr>
<td>kXRS_othercreds</td>
<td>3017</td>
<td>Alternative creds (e.g. other crypto)</td>
<td></td>
</tr>
<tr>
<td>Variable</td>
<td>Index</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>------------------</td>
<td>-------</td>
<td>---------------------------------------</td>
<td></td>
</tr>
<tr>
<td>kXRS_cache_idx</td>
<td>3018</td>
<td>Cache entry index</td>
<td></td>
</tr>
<tr>
<td>kXRS_clnt_opts</td>
<td>3019</td>
<td>Client options, if any</td>
<td></td>
</tr>
<tr>
<td>kXRS_error_code</td>
<td>3020</td>
<td>Error code</td>
<td></td>
</tr>
<tr>
<td>kXRS_timestamp</td>
<td>3021</td>
<td>Time stamp</td>
<td></td>
</tr>
<tr>
<td>kXRS_x509</td>
<td>3022</td>
<td>X509 certificate</td>
<td></td>
</tr>
<tr>
<td>kXRS_issuer_hash</td>
<td>3023</td>
<td>Issuer hash</td>
<td></td>
</tr>
<tr>
<td>kXRS_x509_req</td>
<td>3024</td>
<td>X509 certificate request</td>
<td></td>
</tr>
<tr>
<td>kXRS_cipher_alg</td>
<td>3025</td>
<td>Cipher algorithm (list)</td>
<td></td>
</tr>
<tr>
<td>kXRS_md_alg</td>
<td>3026</td>
<td>MD algorithm (list)</td>
<td></td>
</tr>
<tr>
<td>kXRS_afsinfo</td>
<td>3027</td>
<td>AFS information</td>
<td></td>
</tr>
<tr>
<td>kXRS_reserved</td>
<td>3028</td>
<td>Reserved</td>
<td></td>
</tr>
</tbody>
</table>
Appendix C - Versions prior to 10400 / v4.9

Server host name verification

The way XrdSecProtocolgsi handles this changed in XRootD v4.9. Before v4.9 the client relied on the DNS to de-alias the hostname and compares this with the common name found in the server certificate DN. SANs were ignored.

Transmission of DH parameters

Before v4.9 (internal GSI version 10400) the DH parameters were transmitted unsigned in a bucket of type kXRS_puk. Processing of the related steps on client and server side are described below.

Client: step kXGS_cert processing

Received buffer

The information is contained in both the global and main buffers.

Actions performed

The client performs the following actions:

9. Check the cached timestamp against the current timestamp; allow for 300 seconds skew;
10. Get from the global buffer the bucket of type kXRS_cipher_alg with the cipher algorithm list supported by the server; chosen the first one supported locally; update the bucket with the name of the chosen algorithm;
11. Get from the global buffer the bucket of type kXRS_puk with the server public key for DH key agreement, initialize the session cipher and store it in sessionKey;
   a. Drop bucket kXRS_puk from the global buffer;
12. Get from the global buffer the bucket of type kXRS_x509 with the server certificate and used it to finalize the server certificate chain; verify the chain validity.
   a. Drop bucket kXRS_x509 from the global buffer;
13. Verify the server identity: check the server hostname against the certificate Distinguished Name (DN) and, possibly, the Alternative names;
14. Extract the public key from the server certificate and save it in sessionKver;
15. Get from the global buffer the bucket of type kXRS_md_alg with the message digest algorithm list supported by the server; chosen the first one supported locally; update the bucket with the name of the chosen algorithm
16. Get from the global buffer the bucket of type kXRS_main and deserialize it.
Preparation of the reply to kXGS_cert

The following information is added to the global buffer bpar:
8. A bucket of type kXRS_puk with the client public part of the cipher;
9. A bucket of type kXRS_x509 with the client proxy certificate;
10. A bucket of type kXRS_user with the name of the user;

The client sets the next step, internally nextstep, for the server to be kXGC_cert.

Server: step kXGC_cert processing

Received buffer

The information is contained in both the global and main buffers.

Actions performed

The server performs the following actions:
13. Check the cached timestamp against the current timestamp; allow for 300 seconds skew;
14. Get from the global buffer the bucket of type kXRS_cipher_alg with the cipher algorithm list supported by the server; chosen the first one supported locally; update the bucket with the name of the chosen algorithm.
15. Get from the global buffer the bucket of type kXRS_puk with the server public part for session cipher initialize the session cipher and store it in sessionKey;
   a. Drop bucket kXRS_puk from the global buffer;
16. Extract from the global buffer the bucket of type kXRS_main with the main bucket; decrypt the bucket with sessionKey; deserialize the main buffer;
17. Get from the global buffer the bucket of type kXRS_x509 with the client proxy certificate and used it to finalize the client proxy certificate chain; verify the chain validity.
   a. Drop bucket kXRS_x509 from the global buffer;
18. Verify the server identity: check the server hostname against the certificate Distinguished Name (DN) and, possibly, the Alternative names;
19. Extract the public key from the server certificate and save it in sessionKver;
20. If delegate proxies are requested save the client proxy certificate chain;
21. If a request for delegate proxy certificate is required, prepare it and save it into a bucket of type kXRS_x509_req;
22. Get from the global buffer the bucket of type kXRS_md_alg with the message digest algorithm chosen by the client; load it in sessionMD.
23. If a lookup of the gridmap file is required, check the gridmap file and fill Entity.name with the result; in case of failure, use the DN - or the DN hash, if required;
24. If the extraction of the VOMS attributes is required, call the chosen function and fill the relevant fields in Entity with the result;
25. If authorization is required, run the relevant options.