

Adding support for Globus Proxy Certificates to MySQL

S. Eckmann and A. Vaniachine
2007 June 18

Table of Contents

1. Introduction	1
2. Background.....	1
2.1 Certificate-based authentication on the Grid	1
2.2 Certificate-based authentication in MySQL and OpenSSL.....	2
2.3 Approaches investigated for Grid-enabling MySQL	2
3. Details of the implementation	3
3.1 Software versions	3
3.2 OpenSSL changes.....	3
3.3 MySQL changes	5
4. Testing	5
4.1 Test summary	5
4.2 Test details.....	6
4.2.1 Enabling use of proxy certificates in OpenSSL.....	6
4.2.2 Using certificate-based authentication in MySQL.....	7
4.2.3 A surprising feature of MySQL authentication	8
4.2.4 Aliases, scripts, and configurations that facilitate testing.....	8
5. Bibliography	10
Attachments.....	11

List of Tables

Table 1. Differences between the three Proxy Certificate types.....	1
Table 2. Summary of test results.....	6

1. Introduction

This report summarizes our work under the ANL SHIELDS project to Grid-enable MySQL. By “Grid-enable MySQL” we mean adding the ability to use Globus “legacy” and “pre-RFC” proxy certificates in MySQL authentication. Ultimately we did this by modifying OpenSSL to support Globus proxy certificates, then making only minor changes to MySQL to take advantage of the modified OpenSSL.

Section 2 explains in more detail what the problem was and why a solution was desirable, and also describes the candidate solutions that were investigated. Section 3 provides details of the approach that was successfully implemented. Section 4 presents test results that validate the changes, and includes notes that might be helpful in trying to reproduce the work or use the Grid-enabled MySQL or OpenSSL. A bibliography is in Section 5. Attachments contain all relevant source code files to make the report more self-contained for the reader who is interested in technical details.

2. Background

2.1 Certificate-based authentication on the Grid

X.509 Public Key End Entity (EE) Certificates, defined in RFC 3280, are used for authentication in many applications. The Globus Project pioneered the use of “proxy certificates” in Grid applications to allow delegation of privileges from users to processes running on behalf of those users. The details of proxy certificates were first specified in Globus Security Infrastructure (GSI) 2, then redefined in GSI 3, and finally standardized in RFC 3820.

According to RFC 3820, “the term Proxy Certificate is used to describe a certificate that is derived from, and signed by, a normal X.509 Public Key End Entity Certificate or by another Proxy Certificate for the purpose of providing restricted proxying and delegation within a PKI-based authentication system.”

The differences between RFC3820 proxy certificates and the earlier Globus types are summarized in Table 1. These were identified by first using the `grid-proxy-init` command with the “`-old`” option to make a GSI-2 proxy certificate, with the “`-rfc`” option to make an RFC3820 proxy certificate, and with no option to make a GSI-3 proxy certificate, and then comparing the output of

```
grid-proxy-info -text -f <proxy_cert_file>
```

for each of the three types of proxy certificate. More technical details about the differences are in Section 3.2.

Table 1. Differences between the three Proxy Certificate types

Certificate Type	Second CN in Subject field	Proxy Certificate Extension
GSI-2 (“legacy”)	The literal string “proxy”	none
GSI-3 (“pre-RFC”)	Certificate serial number	“Proxy Certificate Info Extension (old OID)”
RFC3820	Certificate serial number	“Proxy Certificate Info Extension”

The reason the differences are important is that the “Grid”, at least in the High-Energy Physics (HEP) community, uses Globus legacy proxy certificates for authentication. The HEP Grid community also uses MySQL, which, prior to completion of the work described in this report,

did *not* support Globus proxy certificates for authentication. There are two ways this incompatibility might be resolved:

- Modify Grid clients to use RFC3820 proxy certificates, or
- Modify applications such as MySQL to support Globus proxy certificates

The first approach would require a worldwide upgrade of all the Grid software to Globus Toolkit 4 (GT4) from Globus GT3 or even GT2. That isn't feasible now because most Worldwide LHC Computing Grid (WLCG) sites are preparing for the LHC data to start coming in November of this year. Now is *not* a good time for a major Globus version upgrade!

Therefore, it is necessary for MySQL to support Globus legacy and pre-RFC proxy certificates.

2.2 Certificate-based authentication in MySQL and OpenSSL

MySQL uses OpenSSL for its X.509 certificate support, so MySQL supports just the set of certificate types supported by OpenSSL. OpenSSL 0.9.8, the version used in this project, supports RFC3280 End Entity Certificates and RFC3820 Proxy Certificates. When either type of Globus proxy certificate is presented to the OpenSSL `verify` functions, the unknown GSI-2 and GSI-3 proxy certificate types fail the verification checks. This is correct behavior and happens whether the unknown certificate type is presented directly using the “`openssl verify`” command from a shell or using the OpenSSL API from an application such as MySQL.

2.3 Approaches investigated for Grid-enabling MySQL

Four approaches to Grid-enabling MySQL were investigated:

- **Patch MySQL with GSI calls**

This approach was started in 2006 but not completed. This was a desirable approach because the Grid user community already has GSI libraries installed. We found, however, that the GSI C-language API documentation was incomplete and no working examples of its use for authentication were available.

Although this was the first approach tried, it was eventually rejected as offering low probability of success.

- **Patch MySQL with GridSite calls or similar**

GridSite is a collection of libraries and tools for adding X.509 certificate support to standard web browsers. The interesting feature of GridSite for our purpose was that it includes support for Globus proxy certificates. Although we probably would not want to use GridSite libraries directly in MySQL, we might be able to build on the GridSite implementation of Globus proxy certificate support.

This approach was investigated for only a short time before being rejected because it appeared to be harder than, and offer no advantages over, the next approach in this list.

- **Modify OpenSSL to support the two kinds of Globus proxy certificates**

Since MySQL uses OpenSSL for its certificate-based authentication, if OpenSSL were changed to support Globus proxy certificates then MySQL would get that support “for free”.

- **Patch MySQL with custom OpenSSL verify/handshake callbacks**

OpenSSL allows the application developer to write “callback” functions to augment or replace the default verification functions. For example, one might write a `verify` callback that replaces Globus legacy and pre-RFC proxy certificates with “equivalent”

RFC3820 proxy certificates, then passes the modified `x509_STORE_CTX` through to the standard `x509_verify_cert()` function. SSL has a function `ssl_cert_dup()` that would help, then one might only have to insert the right proxy extension in the cloned certificate, and delete the pre-RFC proxy extension if it is there.

This approach was rejected after limited investigation because writing OpenSSL callbacks and related code is complicated, so it makes more sense to use and adapt existing callbacks if possible, as in the other approaches. Furthermore, there are significant changes in the ASN.1 code in OpenSSL between 0.9.7d (used in Globus) and any version of 0.9.8, which is needed for basic proxy certificate support.

The third approach – modifying OpenSSL to support Globus proxy certificates – was ultimately used, although MySQL support turned out to be not quite for free. The implementation is described in the next section.

3. Details of the implementation

The successful implementation is based on identifying where OpenSSL tests for RFC3820 proxy certificates, and extending those checks to include Globus proxy certificates. This required almost 2000 lines of new code in OpenSSL, most of it borrowed from Globus Toolkit 4.0.3.

A significant advantage of Grid-enabling MySQL by modifying OpenSSL is that the modified OpenSSL can be used to Grid-enable other applications that use OpenSSL with little or no change to those applications. For example, the PostgreSQL database application could be Grid-enabled by changing or adding fewer than 20 lines of code, almost identical to the MySQL changes documented below.

3.1 Software versions

The following versions of the major software packages were used:

- OpenSSL 0.9.8e
- MySQL 5.0.37
- Globus Toolkit 4.0.3
- Linux kernel 2.6, 32-bit with associated 32-bit libraries and tools
- gcc 4.1 and associated libraries and tools (gcc 3.4 also was used)

3.2 OpenSSL changes

OpenSSL 0.9.8e was modified to recognize Globus GSI-2 and GSI-3 proxy certificates, and to apply essentially the same verification procedures to them as to RFC3820 proxy certificates. This was done by:

- making changes in two existing OpenSSL files: `crypto/x509/x509_vfy.c` and `crypto/x509v3/v3_purp.c`;
- adding two new files: `crypto/x509/globus_proxycert.c` and `crypto/x509/globus_proxycert.h`, containing both new code and code borrowed from Globus Toolkit 4.0.3;
- copying four files from Globus Toolkit 4.0.3 into OpenSSL's `crypto/x509` directory: `proxycertinfo.c`, `proxycertinfo.h`, `proxypolicy.c`, and `proxypolicy.h`; and
- changing the OpenSSL `crypto/x509` Makefile to use the new and copied files.

The changes in existing OpenSSL files are summarized in “patch” form in Attachment 1 “openssl.globus.patch”. The modified OpenSSL files themselves are in Attachments 4 and 5. New files are in Attachments 6 and 7. Files copied from Globus Toolkit 4.0.3 are in Attachments 8–11.

A summary of the changes and new code follows. All line numbers refer to the modified or new files in the attachments:

- In existing file `x509_vfy.c`:
 - At lines 142–148 in function `x509_verify_cert()`: ensure that the new Globus `proxyCertInfo` object is available, by initializing it if necessary. This is essential because of the way ASN.1 parsing code works. OpenSSL has hard-coded arrays of ASN.1 objects and associated OIDs and NIDs. Object lookups can be by name, by NID, or by OID.
 - At line 425 in function `check_chain_extensions()`, comment out the check for the environment variable `OPENSSL_ALLOW_PROXY_CERTS` that controls whether proxy certificates are allowed, and instead allow their use always. This eliminates the need for a user to set that environment variable, which is reasonable since the whole point of our modifications was to extend OpenSSL’s support of proxy certificates.
 - At lines 442–445 and lines 523–525 in function `check_chain_extensions()`, where there were already proxy certificate checks, add another check for Globus proxy certificates via a call to the `is_old_globus_proxy()` function in `globus_proxycert.c`.
- In existing file `v3_purp.c`:
 - At line 292 in function `x509_supported_extension()`, add the NID for our new `globusProxyCertInfo` object to the list of NIDs recognized by OpenSSL.
 - At lines 361–374 in function `x509v3_cache_extensions()`, add a check for GSI-3 proxy certificates, using code borrowed from Globus Toolkit 4.0.3.
 - At lines 669–671 in function `x509_check_issued()`, add a check for GSI-2 proxy certificates via a call to the `is_old_globus_proxy()` function in `globus_proxycert.c`.
- In new file `globus_proxycert.c`:
 - At lines 420–437 in function `check_globus_init()`, create a `globusProxyCertInfo` extension object, needed for checking NID and OID. Code is borrowed from Globus Toolkit 4.0.3 (the Globus file is cited in a comment).
 - At lines 85–403 in function `get_globus_cert_type()`, using code borrowed from Globus Toolkit 4.0.3 (specific files are cited in comments in `globus_proxycert.c`), compute a certificate’s “type”.
 - At lines 448–469 in function `is_old_globus_proxy()`, based on a certificate’s computed type, inform a caller whether the certificate is a Globus proxy certificate or not.
- The copied files `proxycertinfo.c` and `proxypolicy.c` contain most of the Globus Toolkit code for creating, parsing, and verifying the GSI-3 (pre-RFC3820) proxy certificate extension.

The implementation has at least two minor shortcomings:

- The Globus code for computing a certificate’s type has extensive Globus-specific error reporting, the code for which is merely commented out in the borrowed version. It might be useful to insert standard OpenSSL error reporting. However, doing that cleanly would require adding additional error value constants and other changes outside the small set of source files that we modified. To minimize the set of changed files, we left out the detailed error reporting. The type computation function returns uninterpreted constants for errors. Those constants won’t mean anything without looking at the source code.
- Also to minimize the set of changed files, the NID value for the new `globusProxyCertInfo` object is hard-coded in `globus_proxycert.h`, and the object is created dynamically in `globus_proxycert.c`, instead of adding the object and related definitions and code to files in the `crypto/objects` directory.

Another factor to consider is that our modifications have not been merged back into the OpenSSL baseline, so when that changes – for example, when 0.9.9 is released – the new version will not include support for Globus proxy certificates.

3.3 MySQL changes

Since OpenSSL was modified to support Globus proxy certificates, it would be reasonable to expect that no changes were needed to MySQL itself. However, that wasn’t quite true. Three changes were made in the file `vio/visslfactories.c`, only the first of which was essential to supporting Globus proxy certificates:

- the most important change was that a call to `SSL_CTX_use_certificate_file()` was replaced by a call to `SSL_CTX_use_certificate_chain_file()`. The first function doesn’t properly use the chain of signatures in a proxy certificate file but the second does. The proxy certificate file contains a copy of the EE certificate that signed the proxy certificate, specifically to facilitate signature verification.
- the default passphrase prompt was changed to “Enter GRID pass phrase:” to distinguish the Grid-enabled MySQL from a non-Grid-enabled server.
- the SSL “eNULL” cipher was added to the list of allowed ciphers, so that SSL can be used for authentication without encrypting the communication.

The MySQL changes are presented in Attachment 2 “`mysql.globus.patch`”, and the complete `visslfactories.c` file is in Attachment 3.

4. Testing

4.1 Test summary

Since both OpenSSL and MySQL were modified, both were tested. Testing included regression tests (e.g., behavior for end-entity certificates should not have changed). All EE and proxy certificates were tested when valid, when expired, and when not-yet-valid. This was done by changing the system clock to an hour after the certificate’s “notAfter” time (to make it expired) or to an hour before the certificate’s “notBefore” time (to make the certificate not-yet-valid), using the `cert-expire.pl` and `cert-not-yet-valid.pl` scripts that are included in Attachments 12 and 13. This was done mainly because we noticed that MySQL does not correctly handle expired EE certificates or not-yet-valid RFC-compliant proxy certificates.

Fixing that erroneous behavior was an extra benefit of the changes made to support Globus proxy certificates.

OpenSSL was tested by attempting to verify an end-entity certificate and the three kinds of proxy certificate, using the “`openssl verify`” command.

MySQL was tested by first issuing the `mysql` command:

```
grant all on *.* to 'eckmann'@'localhost' require SSL;
```

then using the `mysql` client to connect to the `mysqld` server, attempting to authenticate using an end-entity certificate or one of the three kinds of proxy certificate.

Test results are summarized in Table 2. The key below the table describes what each cell entry means.

Table 2. Summary of test results

System tested	EE cert			Legacy proxy			Pre-RFC proxy			RFC3820 Proxy		
	<i>not yet valid</i>	<i>valid</i>	<i>exp</i>									
Plain OpenSSL	R	A	R	-	r	-	-	r	-	R	A	R
Modified OpenSSL	R	A	R	R	A	R	R	A	R	R	A	R
Plain MySQL	R	A	a	-	r	-	-	r	-	a	A	R
Modified MySQL	R	A	R	R	A	R	R	A	R	R	A	R
Key:												
A the certificate is accepted, as expected												
a the certificate is accepted but should be rejected												
R the certificate is rejected, as expected												
r the certificate is rejected but should be accepted												
- the certificate is rejected, as expected, but for the wrong reason												

The cells highlighted in yellow indicate erroneous results, as explained in the key. The most important thing to note is that the modified OpenSSL and modified MySQL produced no erroneous results: all certificates were accepted or rejected when they should have been.

The only surprising results are those in the “Plain MySQL” row in which a certificate that should have been rejected was accepted. In the case of the RFC3820 proxy certificate this is due to code in `vio_verify_callback()` that changes a return code from “reject” to “accept” for any certificate with a depth greater than zero. Since a proxy certificate is always at depth at least one (because the signing EE certificate is in the same file), proxy certificates are always accepted by MySQL (as long as they are recognized as proxy certificates). We did not investigate the cause of an expired EE cert being accepted.

4.2 Test details

4.2.1 Enabling use of proxy certificates in OpenSSL

By default, OpenSSL does not allow proxy certificates. The unmodified OpenSSL looks for an environment variable `OPENSSL_ALLOW_PROXY_CERTS` with a non-zero value in order to enable use of RFC3820 proxy certificates. As noted in Section 3.2, our modified OpenSSL includes a

change so that proxy certificates are subject to the verification process instead of being automatically rejected.

4.2.2 Using certificate-based authentication in MySQL

In order to use certificate-based authentication in MySQL, the `mysqld` server must have been built with certificate support enabled, and a server certificate must be installed and available. For our testing we installed into `/etc/grid-security` a server certificate issued by the DOEGrids Certificate Authority.

MySQL doesn't actually have a way to configure the `mysqld` server to require SSL connections for all users; it must be done for each user by manipulating the "grant" tables `mysql.user` and `mysql.db`. The default installation has an empty `mysql.db` table and this `mysql.user` table:

```
mysql> select user,host,password,ssl_type,ssl_cipher from mysql.user;
+-----+-----+-----+-----+-----+
| user | host      | password | ssl_type | ssl_cipher |
+-----+-----+-----+-----+-----+
| root | localhost |          |          |          |
| root | anl       |          |          |          |
|      | anl       |          |          |          |
|      | localhost |          |          |          |
+-----+-----+-----+-----+-----+
```

The fact that the `ssl_type` column is empty implies that certificate-based authentication is not required for either of the default users. For testing our changes, the grant tables were changed to disallow anonymous connections, require a specific EE certificate for user `eckmann`, and require signing by `eckmann`'s EE certificate for MySQL user `h194f8e25` connecting to database `TestLFC`:¹

```
mysql> delete from mysql.user where user='';
mysql> delete from mysql.db where user='';
mysql> create database TestLFC;
mysql> grant all on TestLFC.* to eckmann@'localhost' \
    require SUBJECT '/DC=org/DC=doegrids/OU=People/CN=Steve Eckmann 957509';
mysql> grant all on TestLFC.* to eckmann@'%' \
    require SUBJECT '/DC=org/DC=doegrids/OU=People/CN=Steve Eckmann 957509';
mysql> grant select,insert on TestLFC.* to h194f8e25 \
    require ISSUER '/DC=org/DC=doegrids/OU=People/CN=Steve Eckmann 957509';

mysql> select user,host,ssl_type,x509_subject,x509_issuer from mysql.user;
```

user	host	ssl_type	x509_subject	x509_issuer
root	localhost			
root	anl			
eckmann	%	SPECIFIED	/DC=org/DC=doegrids/OU=People/CN=Steve Eckmann 957509	
eckmann	localhost	SPECIFIED	/DC=org/DC=doegrids/OU=People/CN=Steve	

¹ By convention, we use the hash returned by “`grid-cert-info -ih -f /tmp/x509up_`id -u``” preceded by ‘n’ as the MySQL userid associated with a proxy certificate. For user `eckmann` the hash is `194f8e25`, so the userid is `h194f8e25`.

² The output has been reformatted to better fit on the page without wrapping rows returned by the query.

			Eckmann 957509	
h194f8e25	%	SPECIFIED		/DC=org/DC=doegrids/OU=People/CN=Steve Eckmann 957509

```
mysql> select host,db,user from mysql.db;
+-----+-----+-----+
| host | db   | user |
+-----+-----+-----+
| %    | TestLFC | eckmann |
| %    | TestLFC | h194f8e25 |
| localhost | TestLFC | eckmann |
+-----+-----+-----+
```

Finally, the MySQL server and client must be started with options that tell them where to find certificates. The scripts in Section 4.2.4.2 provide examples of how to do this.

4.2.3 A surprising feature of MySQL authentication

The reason we deleted the grant table rows for anonymous connections is that MySQL authentication has a surprising feature, as explained in this excerpt from the MySQL 5.0 Reference Manual:

It is a common misconception to think that, for a given username, all rows that explicitly name that user are used first when the server attempts to find a match for the connection. This is simply not true. The previous example illustrates this, where a connection from thomas.loc.gov by jeffrey is first matched not by the row containing 'jeffrey' as the User column value, but by the row with no username. As a result, jeffrey is authenticated as an anonymous user, even though he specified a username when connecting.

For example, suppose we have these rows in mysql.user:

HOST	USER	Other fields
localhost	''	Other values [no privileges]
%	eckmann	Other values [all privileges except GRANT]

and no other rows with user=eckmann. Further suppose that mysql.db has a single row:

HOST	USER	DB	Other fields
localhost	''	TestLFC	[all privileges except GRANT]

Then trying to connect like this:

```
mysql --user=eckmann --host=localhost test
```

results in the error: Access denied for ''@'localhost' ...

This happens because MySQL authenticated eckmann using the first matching row in the sorted mysql.user table, which happened to be the row with an anonymous user and no global privileges.

The easiest solution to this surprise is to delete the rows for the anonymous user.

4.2.4 Aliases, scripts, and configurations that facilitate testing

4.2.4.1 Aliases for making and verifying proxy certificates

To test OpenSSL I used several csh aliases:

```

# make the three kinds of proxy certs
alias mko 'grid-proxy-init -old'
alias mkp 'grid-proxy-init'
alias mkr 'grid-proxy-init -rfc'

# verify a certificate
alias v 'openssl verify -CApath /etc/grid-security/certificates/ \
    -verbose -purpose sslclient \!*'
# verify a proxy cert with openssl
alias vo 'mko; v -CAfile $HOME/.globus/usercert.pem /tmp/x509up_u`id -u`'
alias vp 'mkp; v -CAfile $HOME/.globus/usercert.pem /tmp/x509up_u`id -u`'
alias vr 'mkr; v -CAfile $HOME/.globus/usercert.pem /tmp/x509up_u`id -u`'

```

Note that since proxy certificates have a default lifetime of 24 hours, it is necessary to create new proxy certificates regularly in order to properly test validity checking.

4.2.4.2 Scripts for starting the MySQL server and client

To start `mysqld` with SSL enabled I use the following `start-mysqld` script:

```

#!/bin/sh
export GRID_HOME=/etc/grid-security
$MYSQL_HOME/bin/mysqld_safe \
    --ssl-capath=$GRID_HOME/certificates \
    --ssl-cert=$GRID_HOME/hostcert.pem \
    --ssl-key=$GRID_HOME/hostkey.pem \
    --log --log-error --debug \
    --user=mysql \
    $* &

```

To test MySQL connections I use this `mysql-w-EE` script:

```

#! /bin/sh

$MYSQL_HOME/bin/mysql \
    --ssl-capath=/etc/grid-security/certificates \
    --ssl-cert=$HOME/.globus/usercert.pem \
    --ssl-key=$HOME/.globus/userkey.pem \
    $*

```

and this `mysql-w-proxy` script:

```

#! /bin/sh

# This script illustrates how to invoke the mysql client with
# a proxy certificate, assuming standard Grid/Globus locations for
# certificates. It is not necessary to use a Globus-enabled client;
# only the server (mysqld) needs to be aware of Globus proxy certs.

proxy_cert="/tmp/x509up_u`id -u`"
$MYSQL_HOME/bin/mysql \
    --ssl-capath=/etc/grid-security/certificates \
    --ssl-cert=$proxy_cert \
    --ssl-key=$proxy_cert \

```

\$ *

For example:

```
mysql-w-EE      # connect as me using my EE cert
mysql-w-proxy   # connect as me using my Proxy cert
mysql-w-proxy --user h194f8e25 TestLFC
                  # connect to TestLFC as h194f8e25 using my Proxy cert
```

5. Bibliography

- | | |
|-------------|--|
| ASN.1 | http://asn1.elibel.tm.fr/ |
| DOEGrids CA | http://www.doeagrids.org/ |
| Globus | http://www.globus.org/
http://www-unix.globus.org/api/c-globus-4.0/ |
| GridSite | http://www.gridsite.org/ |
| Larmouth00 | John Larmouth, <i>ASN.1 Complete</i> , Morgan Kaufman, 2000. |
| MySQL | http://www.mysql.com/ |
| OpenSSL | http://www.openssl.org/ |
| RFC 3280 | “Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile”, IETF, April 2002.
http://www.ietf.org/rfc/rfc3280.txt |
| RFC 3820 | “Internet X.509 Public Key Infrastructure (PKI) Proxy Certificate Profile”, IETF, June 2004.
http://www.ietf.org/rfc/rfc3820.txt |
| SHIELDS | http://www.hep.anl.gov/atlas/ACG/SHIELDS/ |
| Viega02 | John Viega, Matt Messier, and Pravir Chandra, <i>Network Security with OpenSSL</i> , O'Reilly, 2002. |

Attachments

Several OpenSSL and MySQL source files were referenced in this report, as were “patch” files for OpenSSL and MySQL, and a pair of perl scripts for making certificates “expired” or “not-yet-valid”. These files all are provided as attachments for easy reference.

- Attachment 1** openssl.globus.patch
- Attachment 2** mysql.globus.patch
- Attachment 3** viosslfactories.c
- Attachment 4** x509_vfy.c
- Attachment 5** v3_purp.c
- Attachment 6** globus_proxycert.c
- Attachment 7** globus_proxycert.h
- Attachment 8** proxycertinfo.c
- Attachment 9** proxycertinfo.h
- Attachment 10** proxypolicy.c
- Attachment 11** proxypolicy.h
- Attachment 12** cert-expire.pl
- Attachment 13** cert-not-yet-valid.pl

Attachment 1

`openssl.globus.patch`

```

openssl.globus.patch      Fri Jun 15 16:09:40 2007      1
Only in /home/eckmann/work/mysql-globus-5.0.37/openssl-0.9.8e/crypto/x509: globus_proxycert.c
Only in /home/eckmann/work/mysql-globus-5.0.37/openssl-0.9.8e/crypto/x509: globus_proxycert.h
diff -aur crypto/x509/Makefile /home/eckmann/work/mysql-globus-5.0.37/openssl-0.9.8e/crypto/x509/Makefile
--- crypto/x509/Makefile      2006-02-03 18:49:31.000000000 -0700
+++ /home/eckmann/work/mysql-globus-5.0.37/openssl-0.9.8e/crypto/x509/Makefile  2007-03-27 19:36:41.000000000 -0600
@@ -22,13 +22,15 @@
        x509_set.c x509cset.c x509rset.c x509_err.c \
        x509name.c x509_v3.c x509_ext.c x509_att.c \
        x509type.c x509_lu.c x_all.c x509_txt.c \
-       x509_trs.c by_file.c by_dir.c x509_vpm.c
+       x509_trs.c by_file.c by_dir.c x509_vpm.c \
+       globus_proxycert.c proxycertinfo.c proxypolicy.c
LIBOBJ= x509_def.o x509_d2.o x509_r2x.o x509_cmp.o \
         x509_obj.o x509_req.o x509spki.o x509_vfy.o \
         x509_set.o x509cset.o x509rset.o x509_err.o \
         x509name.o x509_v3.o x509_ext.o x509_att.o \
         x509type.o x509_lu.o x_all.o x509_txt.o \
-       x509_trs.o by_file.o by_dir.o x509_vpm.o
+       x509_trs.o by_file.o by_dir.o x509_vpm.o \
+       globus_proxycert.o proxycertinfo.o proxypolicy.o

SRC= $(LIBSRC)

Only in /home/eckmann/work/mysql-globus-5.0.37/openssl-0.9.8e/crypto/x509: proxycertinfo.c
Only in /home/eckmann/work/mysql-globus-5.0.37/openssl-0.9.8e/crypto/x509: proxycertinfo.h
Only in /home/eckmann/work/mysql-globus-5.0.37/openssl-0.9.8e/crypto/x509: proxypolicy.c
Only in /home/eckmann/work/mysql-globus-5.0.37/openssl-0.9.8e/crypto/x509: proxypolicy.h
diff -aur crypto/x509/x509_vfy.c /home/eckmann/work/mysql-globus-5.0.37/openssl-0.9.8e/crypto/x509/x509_vfy.c
--- crypto/x509/x509_vfy.c      2007-02-06 18:42:51.000000000 -0700
+++ /home/eckmann/work/mysql-globus-5.0.37/openssl-0.9.8e/crypto/x509/x509_vfy.c      2007-04-30 19:22:19.000000000 -0600
0
@@ -69,6 +69,7 @@
#include <openssl/x509.h>
#include <openssl/x509v3.h>
#include <openssl/objects.h>
+#include "globus_proxycert.h"

static int null_callback(int ok,X509_STORE_CTX *e);
static int check_issued(X509_STORE_CTX *ctx, X509 *x, X509 *issuer);
@@ -138,6 +139,13 @@
        x=sk_X509_value(ctx->chain,num-1);
        depth=param->depth;

+       /* Ensure that the Globus proxyCertInfo object is available */
+       if (!check_globus_init())
+       {
+           /* Error initializing globus proxyCertInfo object. */

```

```

openssl.globus.patch      Fri Jun 15 16:09:40 2007      2
+
+         X509err(X509_F_X509_VERIFY_CERT,X509_V_ERR_APPLICATION_VERIFICATION);
+         goto end;
+
+     }
+
+     for (;;)
+     {
@@ -414,7 +422,7 @@
+
+     /* A hack to keep people who don't want to modify their software
+      happy */
-     if (getenv("OPENSSL_ALLOW_PROXY_CERTS"))
+     /* if (getenv("OPENSSL_ALLOW_PROXY_CERTS")) */
+         allow_proxy_certs = 1;
+
+     /* Check all untrusted certificates */
@@ -431,7 +439,10 @@
+     ok=cb(0,ctx);
+     if (!ok) goto end;
+ }
-
-     if (!allow_proxy_certs && (x->ex_flags & EXFLAG_PROXY))
+     /* Check for "old" proxy cert: Globus legacy or Globus pre-RFC
+      proxy cert. */
+     if (!allow_proxy_certs &&
+         ((x->ex_flags & EXFLAG_PROXY) || is_old_globus_proxy(x)))
+     {
+         ctx->error = X509_V_ERR_PROXY_CERTIFICATES_NOT_ALLOWED;
+         ctx->error_depth = i;
@@ -509,7 +520,9 @@
+         certificate must be another proxy certificate or a EE
+         certificate. If not, the next certificate must be a
+         CA certificate. */
-
-     if (x->ex_flags & EXFLAG_PROXY)
+     /* Check for "old" proxy cert: Globus legacy or Globus pre-RFC
+      proxy cert. */
+     if ((x->ex_flags & EXFLAG_PROXY) || is_old_globus_proxy(x))
+     {
+         if (x->ex_pcpthlen != -1 && i > x->ex_pcpthlen)
+         {
diff -aur crypto/x509v3/v3_purp.c /home/eckmann/work/mysql-globus-5.0.37/openssl-0.9.8e/crypto/x509v3/v3_purp.c
--- crypto/x509v3/v3_purp.c      2006-12-06 06:38:59.000000000 -0700
+++ /home/eckmann/work/mysql-globus-5.0.37/openssl-0.9.8e/crypto/x509v3/v3_purp.c      2007-04-30 19:22:42.000000000 -060
0
@@ -58,6 +58,7 @@
+
#include <stdio.h>
#include "cryptlib.h"
+#include "x509/globus_proxycert.h"

```

```
#include <openssl/x509v3.h>
#include <openssl/x509_vfy.h>

@@ -291,7 +292,8 @@
        NID_sbgp_ipAddrBlock, /* 290 */
        NID_sbgp_autonomousSysNum, /* 291 */
#endif
-        NID_proxyCertInfo /* 661 */
+        NID_proxyCertInfo, /* 663 */
+        NID_globusProxyCertInfo /* 772 */
    };

    int ex_nid;
@@ -313,6 +315,7 @@
{
    BASIC_CONSTRAINTS *bs;
    PROXY_CERT_INFO_EXTENSION *pci;
+    PROXYCERTINFO *pci_old;
    ASN1_BIT_STRING *usage;
    ASN1_BIT_STRING *ns;
    EXTENDED_KEY_USAGE *extusage;
@@ -341,7 +344,7 @@
        BASIC_CONSTRAINTS_free(bs);
        x->ex_flags |= EXFLAG_BCONS;
    }
-    /* Handle proxy certificates */
+    /* Handle RFC3820 proxy certificates */
    if((pci=X509_get_ext_d2i(x, NID_proxyCertInfo, NULL, NULL))) {
        if (x->ex_flags & EXFLAG_CA
            || X509_get_ext_by_NID(x, NID_subject_alt_name, 0) >= 0
@@ -355,6 +358,20 @@
        PROXY_CERT_INFO_EXTENSION_free(pci);
        x->ex_flags |= EXFLAG_PROXY;
    }
+    /* Handle Globus GSI-3 proxy certificates */
+    if((pci_old=X509_get_ext_d2i(x, NID_globusProxyCertInfo, NULL, NULL))) {
+        if (x->ex_flags & EXFLAG_CA
+            || X509_get_ext_by_NID(x, NID_subject_alt_name, 0) >= 0
+            || X509_get_ext_by_NID(x, NID_issuer_alt_name, 0) >= 0) {
+            x->ex_flags |= EXFLAG_INVALID;
+
+        if (pci_old->path_length) {
+            x->ex_pcpthlen =
+                ASN1_INTEGER_get(pci_old->path_length);
+        } else x->ex_pcpthlen = -1;
+        PROXYCERTINFO_free(pci_old);
+        x->ex_flags |= EXFLAG_PROXY;
+
```

```
+     }
+     /* Handle key usage */
+     if((usage=X509_get_ext_d2i(x, NID_key_usage, NULL, NULL))) {
+         if(usage->length > 0) {
@@ -649,7 +666,9 @@
+             return X509_V_ERR_AKID_ISSUER_SERIAL_MISMATCH;
+         }
}
-     if(subject->ex_flags & EXFLAG_PROXY)
+     /* Check for "old" proxy cert: Globus legacy or Globus pre-RFC
+        proxy cert. */
+     if((subject->ex_flags & EXFLAG_PROXY) || is_old_globus_proxy(subject))
{
    if(ku_reject(issuer, KU_DIGITAL_SIGNATURE))
        return X509_V_ERR_KEYUSAGE_NO_DIGITAL_SIGNATURE;
```

Attachment 2

`mysql.globus.patch`

```
mysql.globus.patch      Fri Jun 15 16:46:18 2007      1
diff -aur vio/visslfactories.c /home/eckmann/work/mysql-globus-5.0.37/mysql-5.0.37/vio/visslfactories.c
--- vio/visslfactories.c      2007-03-05 12:21:22.000000000 -0700
+++ /home/eckmann/work/mysql-globus-5.0.37/mysql-5.0.37/vio/visslfactories.c 2007-04-10 17:02:01.000000000 -0600
@@ -82,7 +82,8 @@
                (long) ctx, cert_file, key_file));
    if (cert_file)
    {
-     if (SSL_CTX_use_certificate_file(ctx, cert_file, SSL_FILETYPE_PEM) <= 0)
+     EVP_set_pw_prompt("Enter GRID pass phrase:");
+     if (SSL_CTX_use_certificate_chain_file(ctx, cert_file) <= 0)
    {
        DBUG_PRINT("error", ("unable to get certificate from '%s'", cert_file));
        DBUG_EXECUTE("error", ERR_print_errors_fp(DBUG_FILE));
@@ -249,8 +250,18 @@
        DBUG_RETURN(0);
    }

+ /* Add the eNULL cipher that can be used */
+ if ( ( SSL_CTX_set_cipher_list(ssl_fd->ssl_context, "ALL:eNULL") == 0 ) )
+
{   DBUG_PRINT("error", ("failed to add eNULL cipher to use"));
+ report_errors();
+ SSL_CTX_free(ssl_fd->ssl_context);
+ my_free((void*)ssl_fd, MYF(0));
+ DBUG_RETURN(0);
}
+
/* Set the ciphers that can be used */
- if (cipher && SSL_CTX_set_cipher_list(ssl_fd->ssl_context, cipher))
+ if (cipher && ( SSL_CTX_set_cipher_list(ssl_fd->ssl_context, cipher) == 0 ) )
{
    DBUG_PRINT("error", ("failed to set ciphers to use"));
    report_errors();
}
```

Attachment 3

viosslfactories.c

```
1: /* Copyright (C) 2000 MySQL AB
2:
3: This program is free software; you can redistribute it and/or modify
4: it under the terms of the GNU General Public License as published by
5: the Free Software Foundation; version 2 of the License.
6:
7: This program is distributed in the hope that it will be useful,
8: but WITHOUT ANY WARRANTY; without even the implied warranty of
9: MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
10: GNU General Public License for more details.
11:
12: You should have received a copy of the GNU General Public License
13: along with this program; if not, write to the Free Software
14: Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA */
15:
16: #include "vio_priv.h"
17:
18: #ifdef HAVE_OPENSSL
19:
20: static bool      ssl_algorithms_added      = FALSE;
21: static bool      ssl_error_strings_loaded= FALSE;
22: static int       verify_depth = 0;
23:
24: static unsigned char dh512_p[ ]=
25: {
26:   0xDA,0x58,0x3C,0x16,0xD9,0x85,0x22,0x89,0xD0,0xE4,0xAF,0x75,
27:   0x6F,0x4C,0xCA,0x92,0xDD,0x4B,0xE5,0x33,0xB8,0x04,0xFB,0x0F,
28:   0xED,0x94,0xEF,0x9C,0x8A,0x44,0x03,0xED,0x57,0x46,0x50,0xD3,
29:   0x69,0x99,0xDB,0x29,0xD7,0x76,0x27,0x6B,0xA2,0xD3,0xD4,0x12,
30:   0xE2,0x18,0xF4,0xDD,0x1E,0x08,0x4C,0xF6,0xD8,0x00,0x3E,0x7C,
31:   0x47,0x74,0xE8,0x33,
32: };
33:
34: static unsigned char dh512_g[ ]={
35:   0x02,
36: };
37:
38: static DH *get_dh512(void)
39: {
40:   DH *dh;
41:   if ((dh=DH_new()))
42:   {
43:     dh->p=BN_bin2bn(dh512_p,sizeof(dh512_p),NULL);
44:     dh->g=BN_bin2bn(dh512_g,sizeof(dh512_g),NULL);
45:     if (! dh->p || ! dh->g)
46:     {
47:       DH_free(dh);
48:       dh=0;
49:     }
50:   }
51:   return(dh);
52: }
53:
54:
55: static void
56: report_errors()
57: {
58:   unsigned long l;
59:   const char* file;
60:   const char* data;
61:   int         line,flags;
62:
63:   DBUG_ENTER("report_errors");
64:
65:   while ((l=ERR_get_error_line_data(&file,&line,&data,&flags)) != 0)
```

```
66:     {
67: #ifndef DBUG_OFF                                /* Avoid warning */
68:     char buf[200];
69:     DBUG_PRINT("error", ("OpenSSL: %s:%s:%d:%s\n", ERR_error_string(l,buf),
70:                         file,line,(flags & ERR_TXT_STRING) ? data : ""));
71: #endif
72:     }
73:     DBUG_VOID_RETURN;
74: }
75:
76:
77: static int
78: vio_set_cert_stuff(SSL_CTX *ctx, const char *cert_file, const char *key_file)
79: {
80:     DBUG_ENTER("vio_set_cert_stuff");
81:     DBUG_PRINT("enter", ("ctx: 0x%lx cert_file: %s key_file: %s",
82:                           (long) ctx, cert_file, key_file));
83:     if (cert_file)
84:     {
85:         EVP_set_pw_prompt("Enter GRID pass phrase:");
86:         if (SSL_CTX_use_certificate_chain_file(ctx,cert_file) <= 0)
87:         {
88:             DBUG_PRINT("error",("unable to get certificate from '%s'", cert_file));
89:             DBUG_EXECUTE("error", ERR_print_errors_fp(DBUG_FILE));
90:             fprintf(stderr, "SSL error: Unable to get certificate from '%s'\n",
91:                     cert_file);
92:             fflush(stderr);
93:             DBUG_RETURN(1);
94:         }
95:
96:         if (!key_file)
97:             key_file= cert_file;
98:
99:         if (SSL_CTX_use_PrivateKey_file(ctx, key_file, SSL_FILETYPE_PEM) <= 0)
100:        {
101:            DBUG_PRINT("error", ("unable to get private key from '%s'", key_file));
102:            DBUG_EXECUTE("error", ERR_print_errors_fp(DBUG_FILE));
103:            fprintf(stderr, "SSL error: Unable to get private key from '%s'\n",
104:                    key_file);
105:            fflush(stderr);
106:            DBUG_RETURN(1);
107:        }
108:
109: /*
110:     If we are using DSA, we can copy the parameters from the private key
111:     Now we know that a key and cert have been set against the SSL context
112: */
113: if (!SSL_CTX_check_private_key(ctx))
114: {
115:     DBUG_PRINT("error",
116:               ("Private key does not match the certificate public key"));
117:     DBUG_EXECUTE("error", ERR_print_errors_fp(DBUG_FILE));
118:     fprintf(stderr,
119:             "SSL error: "
120:             "Private key does not match the certificate public key\n");
121:     fflush(stderr);
122:     DBUG_RETURN(1);
123: }
124: }
125: DBUG_RETURN(0);
126: }
127:
128:
129: static int
130: vio_verify_callback(int ok, X509_STORE_CTX *ctx)
```

```
131: {
132:     char buf[256];
133:     X509 *err_cert;
134:
135:     DBUG_ENTER("vio_verify_callback");
136:     DBUG_PRINT("enter", ("ok: %d  ctx: 0x%lx", ok, (long) ctx));
137:
138:     err_cert= X509_STORE_CTX_get_current_cert(ctx);
139:     X509_NAME_oneline(X509_get_subject_name(err_cert), buf, sizeof(buf));
140:     DBUG_PRINT("info", ("cert: %s", buf));
141:     if (!ok)
142:     {
143:         int err, depth;
144:         err= X509_STORE_CTX_get_error(ctx);
145:         depth= X509_STORE_CTX_get_error_depth(ctx);
146:
147:         DBUG_PRINT("error", ("verify error: %d  '%s'", err,
148:                             X509_verify_cert_error_string(err)));
149:         /*
150:             Approve cert if depth is greater then "verify_depth", currently
151:             verify_depth is always 0 and there is no way to increase it.
152:         */
153:         if (verify_depth >= depth)
154:             ok= 1;
155:     }
156:     switch (ctx->error)
157:     {
158:     case X509_V_ERR_UNABLE_TO_GET_ISSUER_CERT:
159:         X509_NAME_oneline(X509_get_issuer_name(ctx->current_cert), buf, 256);
160:         DBUG_PRINT("info", ("issuer= %s\n", buf));
161:         break;
162:     case X509_V_ERR_CERT_NOT_YET_VALID:
163:     case X509_V_ERR_ERROR_IN_CERT_NOT_BEFORE_FIELD:
164:         DBUG_PRINT("error", ("notBefore"));
165:         /*ASN1_TIME_print_fp(stderr,X509_get_notBefore(ctx->current_cert));*/
166:         break;
167:     case X509_V_ERR_CERT_HAS_EXPIRED:
168:     case X509_V_ERR_ERROR_IN_CERT_NOT_AFTER_FIELD:
169:         DBUG_PRINT("error", ("notAfter error"));
170:         /*ASN1_TIME_print_fp(stderr,X509_get_notAfter(ctx->current_cert));*/
171:         break;
172:     }
173:     DBUG_PRINT("exit", ("%d", ok));
174:     DBUG_RETURN(ok);
175: }
176:
177:
178: #ifdef __NETWARE__
179:
180: /* NetWare SSL cleanup */
181: void netware_ssl_cleanup()
182: {
183:     /* free memory from SSL_library_init() */
184:     EVP_cleanup();
185:
186:     /* OpenSSL NetWare port specific functions */
187: #ifndef HAVE_YASSL
188:
189:     /* free global X509 method */
190:     X509_STORE_method_cleanup();
191:
192:     /* free the thread_hash error table */
193:     ERR_free_state_table();
194: #endif
195: }
```

```
196:
197:
198: /* NetWare SSL initialization */
199: static void netware_ssl_init()
200: {
201:     /* cleanup OpenSSL library */
202:     NXVmRegisterExitHandler(netware_ssl_cleanup, NULL);
203: }
204:
205: #endif /* __NETWARE__ */
206:
207:
208: static void check_ssl_init()
209: {
210:     if (!ssl_algorithms_added)
211:     {
212:         ssl_algorithms_added= TRUE;
213:         SSL_library_init();
214:         OpenSSL_add_all_algorithms();
215:
216:     }
217:
218: #ifdef __NETWARE__
219:     netware_ssl_init();
220: #endif
221:
222:     if (!ssl_error_strings_loaded)
223:     {
224:         ssl_error_strings_loaded= TRUE;
225:         SSL_load_error_strings();
226:     }
227: }
228:
229: /***** VioSSLFd *****/
230: static struct st_VioSSLFd *
231: new_VioSSLFd(const char *key_file, const char *cert_file,
232:               const char *ca_file, const char *ca_path,
233:               const char *cipher, SSL_METHOD *method)
234: {
235:     DH *dh;
236:     struct st_VioSSLFd *ssl_fd;
237:     DBUG_ENTER("new_VioSSLFd");
238:
239:     check_ssl_init();
240:
241:     if (!(ssl_fd= ((struct st_VioSSLFd*)
242:                     my_malloc(sizeof(struct st_VioSSLFd), MYF(0))))))
243:         DBUG_RETURN(0);
244:
245:     if (!(ssl_fd->ssl_context= SSL_CTX_new(method)))
246:     {
247:         DBUG_PRINT("error", ("SSL_CTX_new failed"));
248:         report_errors();
249:         my_free((void*)ssl_fd, MYF(0));
250:         DBUG_RETURN(0);
251:     }
252:
253:     /* Add the eNULL cipher that can be used */
254:     if ( ( SSL_CTX_set_cipher_list(ssl_fd->ssl_context, "ALL:eNULL") == 0 ) )
255:     {
256:         DBUG_PRINT("error", ("failed to add eNULL cipher to use"));
257:         report_errors();
258:         SSL_CTX_free(ssl_fd->ssl_context);
259:         my_free((void*)ssl_fd, MYF(0));
260:         DBUG_RETURN(0);
```

```
261:     }
262:
263:     /* Set the ciphers that can be used */
264:     if (cipher && ( SSL_CTX_set_cipher_list(ssl_fd->ssl_context, cipher) == 0 ) )
265:     {
266:         DBUG_PRINT("error", ("failed to set ciphers to use"));
267:         report_errors();
268:         SSL_CTX_free(ssl_fd->ssl_context);
269:         my_free((void*)ssl_fd,MYF(0));
270:         DBUG_RETURN(0);
271:     }
272:
273:     /* Load certs from the trusted ca */
274:     if (SSL_CTX_load_verify_locations(ssl_fd->ssl_context, ca_file, ca_path) == 0)
275:     {
276:         DBUG_PRINT("warning", ("SSL_CTX_load_verify_locations failed"));
277:         if (SSL_CTX_set_default_verify_paths(ssl_fd->ssl_context) == 0)
278:         {
279:             DBUG_PRINT("error", ("SSL_CTX_set_default_verify_paths failed"));
280:             report_errors();
281:             SSL_CTX_free(ssl_fd->ssl_context);
282:             my_free((void*)ssl_fd,MYF(0));
283:             DBUG_RETURN(0);
284:         }
285:     }
286:
287:     if (vio_set_cert_stuff(ssl_fd->ssl_context, cert_file, key_file))
288:     {
289:         DBUG_PRINT("error", ("vio_set_cert_stuff failed"));
290:         report_errors();
291:         SSL_CTX_free(ssl_fd->ssl_context);
292:         my_free((void*)ssl_fd,MYF(0));
293:         DBUG_RETURN(0);
294:     }
295:
296:     /* DH stuff */
297:     dh=get_dh512();
298:     SSL_CTX_set_tmp_dh(ssl_fd->ssl_context, dh);
299:     DH_free(dh);
300:
301:     DBUG_PRINT("exit", ("OK 1"));
302:
303:     DBUG_RETURN(ssl_fd);
304: }
305:
306:
307: /***** VioSSLConnectorFd *****/
308: struct st_VioSSLFd *
309: new_VioSSLConnectorFd(const char *key_file, const char *cert_file,
310:                       const char *ca_file, const char *ca_path,
311:                       const char *cipher)
312: {
313:     struct st_VioSSLFd *ssl_fd;
314:     int verify= SSL_VERIFY_PEER;
315:     if (!(ssl_fd= new_VioSSLFd(key_file, cert_file, ca_file,
316:                                 ca_path, cipher, TLSv1_client_method())))
317:     {
318:         return 0;
319:     }
320:
321:     /* Init the VioSSLFd as a "connector" ie. the client side */
322:
323:     /*
324:      The verify_callback function is used to control the behaviour
325:      when the SSL_VERIFY_PEER flag is set.
```

```
326:  */
327:  SSL_CTX_set_verify(ssl_fd->ssl_context, verify, vio_verify_callback);
328:
329:  return ssl_fd;
330: }
331:
332:
333: /***** VioSSLAcceptorFd *****/
334: struct st_VioSSLFd*
335: new_VioSSLAcceptorFd(const char *key_file, const char *cert_file,
336:                        const char *ca_file, const char *ca_path,
337:                        const char *cipher)
338: {
339:  struct st_VioSSLFd *ssl_fd;
340:  int verify= SSL_VERIFY_PEER | SSL_VERIFY_CLIENT_ONCE;
341:  if (!(ssl_fd= new_VioSSLFd(key_file, cert_file, ca_file,
342:                             ca_path, cipher, TLSv1_server_method())))
343:  {
344:   return 0;
345:  }
346: /* Init the the VioSSLFd as a "acceptor" ie. the server side */
347:
348: /* Set max number of cached sessions, returns the previous size */
349: SSL_CTX_sess_set_cache_size(ssl_fd->ssl_context, 128);
350:
351: /*
352:   The verify_callback function is used to control the behaviour
353:   when the SSL_VERIFY_PEER flag is set.
354: */
355: SSL_CTX_set_verify(ssl_fd->ssl_context, verify, vio_verify_callback);
356:
357: /*
358:   Set session_id - an identifier for this server session
359:   Use the ssl_fd pointer
360: */
361: SSL_CTX_set_session_id_context(ssl_fd->ssl_context,
362:                                (const unsigned char *)ssl_fd,
363:                                sizeof(ssl_fd));
364:
365: return ssl_fd;
366: }
367: #endif /* HAVE_OPENSSL */
```

Attachment 4

x509_vfy.c

```
1: /* crypto/x509/x509_vfy.c */
2: /* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
3: * All rights reserved.
4: *
5: * This package is an SSL implementation written
6: * by Eric Young (eay@cryptsoft.com).
7: * The implementation was written so as to conform with Netscapes SSL.
8: *
9: * This library is free for commercial and non-commercial use as long as
10: * the following conditions are aheared to. The following conditions
11: * apply to all code found in this distribution, be it the RC4, RSA,
12: * lhash, DES, etc., code; not just the SSL code. The SSL documentation
13: * included with this distribution is covered by the same copyright terms
14: * except that the holder is Tim Hudson (tjh@cryptsoft.com).
15: *
16: * Copyright remains Eric Young's, and as such any Copyright notices in
17: * the code are not to be removed.
18: * If this package is used in a product, Eric Young should be given attribution
19: * as the author of the parts of the library used.
20: * This can be in the form of a textual message at program startup or
21: * in documentation (online or textual) provided with the package.
22: *
23: * Redistribution and use in source and binary forms, with or without
24: * modification, are permitted provided that the following conditions
25: * are met:
26: * 1. Redistributions of source code must retain the copyright
27: * notice, this list of conditions and the following disclaimer.
28: * 2. Redistributions in binary form must reproduce the above copyright
29: * notice, this list of conditions and the following disclaimer in the
30: * documentation and/or other materials provided with the distribution.
31: * 3. All advertising materials mentioning features or use of this software
32: * must display the following acknowledgement:
33: * "This product includes cryptographic software written by
34: * Eric Young (eay@cryptsoft.com)"
35: * The word 'cryptographic' can be left out if the rouines from the library
36: * being used are not cryptographic related :).
37: * 4. If you include any Windows specific code (or a derivative thereof) from
38: * the apps directory (application code) you must include an acknowledgement:
39: * "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
40: *
41: * THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ''AS IS'' AND
42: * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
43: * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
44: * ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
45: * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
46: * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
47: * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
48: * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
49: * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
50: * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
51: * SUCH DAMAGE.
52: *
53: * The licence and distribution terms for any publically available version or
54: * derivative of this code cannot be changed. i.e. this code cannot simply be
55: * copied and put under another distribution licence
56: * [including the GNU Public Licence.]
```

```
66: #include <openssl/buffer.h>
67: #include <openssl/evp.h>
68: #include <openssl/asn1.h>
69: #include <openssl/x509.h>
70: #include <openssl/x509v3.h>
71: #include <openssl/objects.h>
72: #include "globus_proxycert.h"
73:
74: static int null_callback(int ok,X509_STORE_CTX *e);
75: static int check_issued(X509_STORE_CTX *ctx, X509 *x, X509 *issuer);
76: static X509 *find_issuer(X509_STORE_CTX *ctx, STACK_OF(X509) *sk, X509 *x);
77: static int check_chain_extensions(X509_STORE_CTX *ctx);
78: static int check_trust(X509_STORE_CTX *ctx);
79: static int check_revocation(X509_STORE_CTX *ctx);
80: static int check_cert(X509_STORE_CTX *ctx);
81: static int check_policy(X509_STORE_CTX *ctx);
82: static int internal_verify(X509_STORE_CTX *ctx);
83: const char X509_version[ ]="X.509" OPENSSL_VERSION_PTEXT;
84:
85:
86: static int null_callback(int ok, X509_STORE_CTX *e)
87: {
88:     return ok;
89: }
90:
91: #if 0
92: static int x509_subject_cmp(X509 **a, X509 **b)
93: {
94:     return X509_subject_name_cmp(*a,*b);
95: }
96: #endif
97:
98: int X509_verify_cert(X509_STORE_CTX *ctx)
99: {
100:     X509 *x,*xtmp,*chain_ss=NULL;
101:     X509_NAME *xn;
102:     int bad_chain = 0;
103:     X509_VERIFY_PARAM *param = ctx->param;
104:     int depth,i,ok=0;
105:     int num;
106:     int (*cb)(int xok,X509_STORE_CTX *xctx);
107:     STACK_OF(X509) *sktmp=NULL;
108:     if (ctx->cert == NULL)
109:     {
110:         X509err(X509_F_X509_VERIFY_CERT,X509_R_NO_CERT_SET_FOR_US_TO_VERYFY);
111:         return -1;
112:     }
113:
114:     cb=ctx->verify_cb;
115:
116: /* first we make sure the chain we are going to build is
117:  * present and that the first entry is in place */
118: if (ctx->chain == NULL)
119: {
120:     if ((ctx->chain=sk_X509_new_null()) == NULL) ||
121:         (!sk_X509_push(ctx->chain,ctx->cert)))
122:     {
123:         X509err(X509_F_X509_VERIFY_CERT,ERR_R_MALLOC_FAILURE);
124:         goto end;
125:     }
126:     CRYPTO_add(&ctx->cert->references,1,CRYPTO_LOCK_X509);
127:     ctx->last_untrusted=1;
128: }
129:
```

```
130: /* We use a temporary STACK so we can chop and hack at it */
131: if (ctx->untrusted != NULL
132:     && (sktmp=sk_X509_dup(ctx->untrusted)) == NULL)
133: {
134:     X509err(X509_F_X509_VERIFY_CERT,ERR_R_MALLOC_FAILURE);
135:     goto end;
136: }
137:
138: num=sk_X509_num(ctx->chain);
139: x=sk_X509_value(ctx->chain,num-1);
140: depth=param->depth;
141:
142: /* Ensure that the Globus proxyCertInfo object is available */
143: if (!check_globus_init())
144: {
145:     /* Error initializing globus proxyCertInfo object. */
146:     X509err(X509_F_X509_VERIFY_CERT,X509_V_ERR_APPLICATION_VERIFICATION);
147:     goto end;
148: }
149:
150: for (;;)
151: {
152:     /* If we have enough, we break */
153:     if (depth < num) break; /* FIXME: If this happens, we should take
e
154:                                     * note of it and, if appropriate, use t
he
155:                                     * X509_V_ERR_CERT_CHAIN_TOO_LONG error
156:                                     * code later.
157:                                     */
158:
159:     /* If we are self signed, we break */
160:     xn=X509_get_issuer_name(x);
161:     if (ctx->check_issued(ctx, x,x)) break;
162:
163:     /* If we were passed a cert chain, use it first */
164:     if (ctx->untrusted != NULL)
165:     {
166:         xtmp=find_issuer(ctx, sktmp,x);
167:         if (xtmp != NULL)
168:         {
169:             if (!sk_X509_push(ctx->chain,xtmp))
170:             {
171:                 X509err(X509_F_X509_VERIFY_CERT,ERR_R_MALLOC_FAILURE);
172:                 goto end;
173:             }
174:             CRYPTO_add(&xtmp->references,1,CRYPTO_LOCK_X509);
175:             sk_X509_delete_ptr(sktmp,xtmp);
176:             ctx->last_untrusted++;
177:             x=xtmp;
178:             num++;
179:             /* reparse the full chain for
180:                * the next one */
181:             continue;
182:         }
183:     }
184:     break;
185: }
186:
187: /* at this point, chain should contain a list of untrusted
188:    * certificates. We now need to add at least one trusted one,
189:    * if possible, otherwise we complain. */
190:
```

```
191: /* Examine last certificate in chain and see if it
192: * is self signed.
193: */
194:
195: i=sk_X509_num(ctx->chain);
196: x=sk_X509_value(ctx->chain,i-1);
197: xn = X509_get_subject_name(x);
198: if (ctx->check_issued(ctx, x, x))
199: {
200:     /* we have a self signed certificate */
201:     if (sk_X509_num(ctx->chain) == 1)
202:     {
203:         /* We have a single self signed certificate: see if
204:          * we can find it in the store. We must have an exact
205:          * match to avoid possible impersonation.
206:         */
207:         ok = ctx->get_issuer(&xtmp, ctx, x);
208:         if ((ok <= 0) || X509_cmp(x, xtmp))
209:         {
210:             ctx->error=X509_V_ERR_DEPTH_ZERO_SELF_SIGNED_CER
T;
211:             ctx->current_cert=x;
212:             ctx->error_depth=i-1;
213:             if (ok == 1) X509_free(xtmp);
214:             bad_chain = 1;
215:             ok=cb(0,ctx);
216:             if (!ok) goto end;
217:         }
218:         else
219:         {
220:             /* We have a match: replace certificate with sto
re version
221:              * so we get any trust settings.
222:              */
223:             X509_free(x);
224:             x = xtmp;
225:             sk_X509_set(ctx->chain, i - 1, x);
226:             ctx->last_untrusted=0;
227:         }
228:     }
229:     else
230:     {
231:         /* extract and save self signed certificate for later us
e */
232:         chain_ss=sk_X509_pop(ctx->chain);
233:         ctx->last_untrusted--;
234:         num--;
235:         x=sk_X509_value(ctx->chain,num-1);
236:     }
237: }
238:
239: /* We now lookup certs from the certificate store */
240: for (;;)
241: {
242:     /* If we have enough, we break */
243:     if (depth < num) break;
244:
245:     /* If we are self signed, we break */
246:     xn=X509_get_issuer_name(x);
247:     if (ctx->check_issued(ctx,x,x)) break;
248:
249:     ok = ctx->get_issuer(&xtmp, ctx, x);
250:
251:     if (ok < 0) return ok;
252:     if (ok == 0) break;
```

```
253:             x = xtmp;
254:             if (!sk_X509_push(ctx->chain,x))
255:             {
256:                 X509_free(xtmp);
257:                 X509err(X509_F_X509_VERIFY_CERT,ERR_R_MALLOC_FAILURE);
258:                 return 0;
259:             }
260:             num++;
261:         }
262:     }
263:
264:     /* we now have our chain, lets check it... */
265:     xn=X509_get_issuer_name(x);
266:
267:     /* Is last certificate looked up self signed? */
268:     if (!ctx->check_issued(ctx,x,x))
269:     {
270:         if ((chain_ss == NULL) || !ctx->check_issued(ctx, x, chain_ss))
271:         {
272:             if (ctx->last_untrusted >= num)
273:                 ctx->error=X509_V_ERR_UNABLE_TO_GET_ISSUER_CERT_
LOCALLY;
274:
275:             else
276:                 ctx->error=X509_V_ERR_UNABLE_TO_GET_ISSUER_CERT;
277:             ctx->current_cert=x;
278:         }
279:     }
280:
281:     sk_X509_push(ctx->chain,chain_ss);
282:     num++;
283:     ctx->last_untrusted=num;
284:     ctx->current_cert=chain_ss;
285:     ctx->error=X509_V_ERR_SELF_SIGNED_CERT_IN_CHAIN;
286:     chain_ss=NULL;
287: }
288:
289: ctx->error_depth=num-1;
290: bad_chain = 1;
291: ok=cb(0,ctx);
292: if (!ok) goto end;
293: }
294:
295: /* We have the chain complete: now we need to check its purpose */
296: ok = check_chain_extensions(ctx);
297:
298: if (!ok) goto end;
299:
300: /* The chain extensions are OK: check trust */
301:
302: if (param->trust > 0) ok = check_trust(ctx);
303:
304: if (!ok) goto end;
305:
306: /* We may as well copy down any DSA parameters that are required */
307: X509_get_pubkey_parameters(NULL,ctx->chain);
308:
309: /* Check revocation status: we do this after copying parameters
310: * because they may be needed for CRL signature verification.
311: */
312:
313: ok = ctx->check_revocation(ctx);
314: if(!ok) goto end;
315:
316: /* At this point, we have a chain and need to verify it */
```

```

x509_vfy.c      Mon Apr 30 19:22:19 2007      6

317:     if (ctx->verify != NULL)
318:         ok=ctx->verify(ctx);
319:     else
320:         ok=internal_verify(ctx);
321:     if(!ok) goto end;
322:
323: #ifndef OPENSSL_NO_RFC3779
324: /* RFC 3779 path validation, now that CRL check has been done */
325: ok = v3_asid_validate_path(ctx);
326: if (!ok) goto end;
327: ok = v3_addr_validate_path(ctx);
328: if (!ok) goto end;
329: #endif
330:
331: /* If we get this far evaluate policies */
332: if (!bad_chain && (ctx->param->flags & X509_V_FLAG_POLICY_CHECK))
333:     ok = ctx->check_policy(ctx);
334: if(!ok) goto end;
335: if (0)
336: {
337: end:
338:     X509_get_pubkey_parameters(NULL,ctx->chain);
339: }
340: if (sktmp != NULL) sk_X509_free(sktmp);
341: if (chain_ss != NULL) X509_free(chain_ss);
342: return ok;
343: }
344:
345:
346: /* Given a STACK_OF(X509) find the issuer of cert (if any)
347: */
348:
349: static X509 *find_issuer(X509_STORE_CTX *ctx, STACK_OF(X509) *sk, X509 *x)
350: {
351:     int i;
352:     X509 *issuer;
353:     for (i = 0; i < sk_X509_num(sk); i++)
354:     {
355:         issuer = sk_X509_value(sk, i);
356:         if (ctx->check_issued(ctx, x, issuer))
357:             return issuer;
358:     }
359:     return NULL;
360: }
361:
362: /* Given a possible certificate and issuer check them */
363:
364: static int check_issued(X509_STORE_CTX *ctx, X509 *x, X509 *issuer)
365: {
366:     int ret;
367:     ret = X509_check_issued(issuer, x);
368:     if (ret == X509_V_OK)
369:         return 1;
370:     /* If we haven't asked for issuer errors don't set ctx */
371:     if (!(ctx->param->flags & X509_V_FLAG_CB_ISSUER_CHECK))
372:         return 0;
373:
374:     ctx->error = ret;
375:     ctx->current_cert = x;
376:     ctx->current_issuer = issuer;
377:     return ctx->verify_cb(0, ctx);
378:     return 0;
379: }
380:
381: /* Alternative lookup method: look from a STACK stored in other_ctx */

```

```
382:
383: static int get_issuer_sk(X509 **issuer, X509_STORE_CTX *ctx, X509 *x)
384: {
385:     *issuer = find_issuer(ctx, ctx->other_ctx, x);
386:     if (*issuer)
387:     {
388:         CRYPTO_add(&(*issuer)->references, 1, CRYPTO_LOCK_X509);
389:         return 1;
390:     }
391:     else
392:         return 0;
393: }
394:
395:
396: /* Check a certificate chains extensions for consistency
397:  * with the supplied purpose
398: */
399:
400: static int check_chain_extensions(X509_STORE_CTX *ctx)
401: {
402: #ifdef OPENSSL_NO_CHAIN_VERIFY
403:     return 1;
404: #else
405:     int i, ok=0, must_be_ca;
406:     X509 *x;
407:     int (*cb)(int xok,X509_STORE_CTX *xctx);
408:     int proxy_path_length = 0;
409:     int allow_proxy_certs =
410:         !(ctx->param->flags & X509_V_FLAG_ALLOW_PROXY_CERTS);
411:     cb=ctx->verify_cb;
412:
413:     /* must_be_ca can have 1 of 3 values:
414:      -1: we accept both CA and non-CA certificates, to allow direct
415:           use of self-signed certificates (which are marked as CA).
416:          0: we only accept non-CA certificates. This is currently not
417:              used, but the possibility is present for future extensions.
418:          1: we only accept CA certificates. This is currently used for
419:              all certificates in the chain except the leaf certificate.
420:     */
421:     must_be_ca = -1;
422:
423:     /* A hack to keep people who don't want to modify their software
424:      happy */
425:     /* if (getenv("OPENSSL_ALLOW_PROXY_CERTS")) */
426:     allow_proxy_certs = 1;
427:
428:     /* Check all untrusted certificates */
429:     for (i = 0; i < ctx->last_untrusted; i++)
430:     {
431:         int ret;
432:         x = sk_X509_value(ctx->chain, i);
433:         if (!(ctx->param->flags & X509_V_FLAG_IGNORE_CRITICAL)
434:             && (x->ex_flags & EXFLAG_CRITICAL))
435:         {
436:             ctx->error = X509_V_ERR_UNHANDLED_CRITICAL_EXTENSION;
437:             ctx->error_depth = i;
438:             ctx->current_cert = x;
439:             ok=cb(0,ctx);
440:             if (!ok) goto end;
441:         }
442:         /* Check for "old" proxy cert: Globus legacy or Globus pre-RFC
443:            proxy cert. */
444:         if (!allow_proxy_certs &&
445:             ((x->ex_flags & EXFLAG_PROXY) || is_old_globus_proxy(x)))
446:         {
```

```
447:             ctx->error = X509_V_ERR_PROXY_CERTIFICATES_NOT_ALLOWED;
448:             ctx->error_depth = i;
449:             ctx->current_cert = x;
450:             ok=cb(0,ctx);
451:             if (!ok) goto end;
452:         }
453:     ret = X509_check_ca(x);
454:     switch(must_be_ca)
455:     {
456:     case -1:
457:         if (((ctx->param->flags & X509_V_FLAG_X509_STRICT)
458:              && (ret != 1) && (ret != 0))
459:             {
460:             ret = 0;
461:             ctx->error = X509_V_ERR_INVALID_CA;
462:             }
463:         else
464:             ret = 1;
465:         break;
466:     case 0:
467:         if (ret != 0)
468:         {
469:             ret = 0;
470:             ctx->error = X509_V_ERR_INVALID_NON_CA;
471:             }
472:         else
473:             ret = 1;
474:         break;
475:     default:
476:         if ((ret == 0)
477:             || ((ctx->param->flags & X509_V_FLAG_X509_STRICT
)
478:                 && (ret != 1)))
479:         {
480:             ret = 0;
481:             ctx->error = X509_V_ERR_INVALID_CA;
482:             }
483:         else
484:             ret = 1;
485:         break;
486:     }
487:     if (ret == 0)
488:     {
489:         ctx->error_depth = i;
490:         ctx->current_cert = x;
491:         ok=cb(0,ctx);
492:         if (!ok) goto end;
493:     }
494:     if (ctx->param->purpose > 0)
495:     {
496:         ret = X509_check_purpose(x, ctx->param->purpose,
497:                                   must_be_ca > 0);
498:         if ((ret == 0)
499:             || ((ctx->param->flags & X509_V_FLAG_X509_STRICT
)
500:                 && (ret != 1)))
501:         {
502:             ctx->error = X509_V_ERR_INVALID_PURPOSE;
503:             ctx->error_depth = i;
504:             ctx->current_cert = x;
505:             ok=cb(0,ctx);
506:             if (!ok) goto end;
507:         }
508:     }
509: /* Check pathlen */
```

```
510:         if ((i > 1) && (x->ex_pathlen != -1)
511:             && (i > (x->ex_pathlen + proxy_path_length + 1)))
512:             {
513:                 ctx->error = X509_V_ERR_PATH_LENGTH_EXCEEDED;
514:                 ctx->error_depth = i;
515:                 ctx->current_cert = x;
516:                 ok=cb(0,ctx);
517:                 if (!ok) goto end;
518:             }
519:             /* If this certificate is a proxy certificate, the next
520:                certificate must be another proxy certificate or a EE
521:                certificate. If not, the next certificate must be a
522:                CA certificate. */
523:             /* Check for "old" proxy cert: Globus legacy or Globus pre-RFC
524:                proxy cert. */
525:             if ((x->ex_flags & EXFLAG_PROXY) || is_old_globus_proxy(x))
526:             {
527:                 if (x->ex_pcpathlen != -1 && i > x->ex_pcpathlen)
528:                     {
529:                         ctx->error =
530:                             X509_V_ERR_PROXY_PATH_LENGTH_EXCEEDED;
531:                         ctx->error_depth = i;
532:                         ctx->current_cert = x;
533:                         ok=cb(0,ctx);
534:                         if (!ok) goto end;
535:                     }
536:                     proxy_path_length++;
537:                     must_be_ca = 0;
538:                 }
539:             else
540:                 must_be_ca = 1;
541:             }
542:         ok = 1;
543:     end:
544:     return ok;
545: #endif
546: }
547:
548: static int check_trust(X509_STORE_CTX *ctx)
549: {
550: #ifdef OPENSSL_NO_CHAIN_VERIFY
551:     return 1;
552: #else
553:     int i, ok;
554:     X509 *x;
555:     int (*cb)(int xok,X509_STORE_CTX *xctx);
556:     cb=ctx->verify_cb;
557:     /* For now just check the last certificate in the chain */
558:     i = sk_X509_num(ctx->chain) - 1;
559:     x = sk_X509_value(ctx->chain, i);
560:     ok = X509_check_trust(x, ctx->param->trust, 0);
561:     if (ok == X509_TRUST_TRUSTED)
562:         return 1;
563:     ctx->error_depth = i;
564:     ctx->current_cert = x;
565:     if (ok == X509_TRUST_REJECTED)
566:         ctx->error = X509_V_ERR_CERT_REJECTED;
567:     else
568:         ctx->error = X509_V_ERR_CERT_UNTRUSTED;
569:     ok = cb(0, ctx);
570:     return ok;
571: #endif
572: }
573:
574: static int check_revocation(X509_STORE_CTX *ctx)
```

```
575:     {
576:         int i, last, ok;
577:         if (!(ctx->param->flags & X509_V_FLAG_CRL_CHECK))
578:             return 1;
579:         if (ctx->param->flags & X509_V_FLAG_CRL_CHECK_ALL)
580:             last = sk_X509_num(ctx->chain) - 1;
581:         else
582:             last = 0;
583:         for(i = 0; i <= last; i++)
584:         {
585:             ctx->error_depth = i;
586:             ok = check_cert(ctx);
587:             if (!ok) return ok;
588:         }
589:     return 1;
590: }
591:
592: static int check_cert(X509_STORE_CTX *ctx)
593: {
594:     X509_CRL *crl = NULL;
595:     X509 *x;
596:     int ok, cnum;
597:     cnum = ctx->error_depth;
598:     x = sk_X509_value(ctx->chain, cnum);
599:     ctx->current_cert = x;
600:     /* Try to retrieve relevant CRL */
601:     ok = ctx->get_crl(ctx, &crl, x);
602:     /* If error looking up CRL, nothing we can do except
603:      * notify callback
604:      */
605:     if(!ok)
606:     {
607:         ctx->error = X509_V_ERR_UNABLE_TO_GET_CRL;
608:         ok = ctx->verify_cb(0, ctx);
609:         goto err;
610:     }
611:     ctx->current_crl = crl;
612:     ok = ctx->check_crl(ctx, crl);
613:     if (!ok) goto err;
614:     ok = ctx->cert_crl(ctx, crl, x);
615: err:
616:     ctx->current_crl = NULL;
617:     X509_CRL_free(crl);
618:     return ok;
619:
620: }
621:
622: /* Check CRL times against values in X509_STORE_CTX */
623:
624: static int check_crl_time(X509_STORE_CTX *ctx, X509_CRL *crl, int notify)
625: {
626:     time_t *ptime;
627:     int i;
628:     ctx->current_crl = crl;
629:     if (ctx->param->flags & X509_V_FLAG_USE_CHECK_TIME)
630:         ptime = &ctx->param->check_time;
631:     else
632:         ptime = NULL;
633:
634:     i=X509_cmp_time(X509_CRL_get_lastUpdate(crl), ptime);
635:     if (i == 0)
636:     {
637:         ctx->error=X509_V_ERR_ERROR_IN_CRL_LAST_UPDATE_FIELD;
638:         if (!notify || !ctx->verify_cb(0, ctx))
639:             return 0;
```

```

x509_vfy.c      Mon Apr 30 19:22:19 2007      11

640:             }
641:
642:     if (i > 0)
643:     {
644:         ctx->error=X509_V_ERR_CRL_NOT_YET_VALID;
645:         if (!notify || !ctx->verify_cb(0, ctx))
646:             return 0;
647:     }
648:
649:     if(X509_CRL_get_nextUpdate(crl))
650:     {
651:         i=X509_cmp_time(X509_CRL_get_nextUpdate(crl), ptime);
652:
653:         if (i == 0)
654:         {
655:             ctx->error=X509_V_ERR_ERROR_IN_CRL_NEXT_UPDATE_FIELD;
656:             if (!notify || !ctx->verify_cb(0, ctx))
657:                 return 0;
658:         }
659:
660:         if (i < 0)
661:         {
662:             ctx->error=X509_V_ERR_CRL_HAS_EXPIRED;
663:             if (!notify || !ctx->verify_cb(0, ctx))
664:                 return 0;
665:         }
666:     }
667:
668:     ctx->current_crl = NULL;
669:
670:     return 1;
671: }
672:
673: /* Lookup CRLs from the supplied list. Look for matching issuer name
674: * and validity. If we can't find a valid CRL return the last one
675: * with matching name. This gives more meaningful error codes. Otherwise
676: * we'd get a CRL not found error if a CRL existed with matching name but
677: * was invalid.
678: */
679:
680: static int get_crl_sk(X509_STORE_CTX *ctx, X509_CRL **pcrl,
681:                       X509_NAME *nm, STACK_OF(X509_CRL) *crls)
682: {
683:     int i;
684:     X509_CRL *crl, *best_crl = NULL;
685:     for (i = 0; i < sk_X509_CRL_num(crls); i++)
686:     {
687:         crl = sk_X509_CRL_value(crls, i);
688:         if (X509_NAME_cmp(nm, X509_CRL_get_issuer(crl)))
689:             continue;
690:         if (check_crl_time(ctx, crl, 0))
691:         {
692:             *pcrl = crl;
693:             CRYPTO_add(&crl->references, 1, CRYPTO_LOCK_X509);
694:             return 1;
695:         }
696:         best_crl = crl;
697:     }
698:     if (best_crl)
699:     {
700:         *pcrl = best_crl;
701:         CRYPTO_add(&best_crl->references, 1, CRYPTO_LOCK_X509);
702:     }
703:
704:     return 0;

```

```
705:         }
706:
707: /* Retrieve CRL corresponding to certificate: currently just a
708: * subject lookup: maybe use AKID later...
709: */
710: static int get_crl(X509_STORE_CTX *ctx, X509_CRL **pcrl, X509 *x)
711: {
712:     int ok;
713:     X509_CRL *crl = NULL;
714:     X509_OBJECT xobj;
715:     X509_NAME *nm;
716:     nm = X509_get_issuer_name(x);
717:     ok = get_crl_sk(ctx, &crl, nm, ctx->crls);
718:     if (ok)
719:     {
720:         *pcrl = crl;
721:         return 1;
722:     }
723:
724:     ok = X509_STORE_get_by_subject(ctx, X509_LU_CRL, nm, &xobj);
725:
726:     if (!ok)
727:     {
728:         /* If we got a near match from get_crl_sk use that */
729:         if (crl)
730:         {
731:             *pcrl = crl;
732:             return 1;
733:         }
734:         return 0;
735:     }
736:
737:     *pcrl = xobj.data.crl;
738:     if (crl)
739:         X509_CRL_free(crl);
740:     return 1;
741: }
742:
743: /* Check CRL validity */
744: static int check_crl(X509_STORE_CTX *ctx, X509_CRL *crl)
745: {
746:     X509 *issuer = NULL;
747:     EVP_PKEY *ikey = NULL;
748:     int ok = 0, chnum, cnum;
749:     cnum = ctx->error_depth;
750:     chnum = sk_X509_num(ctx->chain) - 1;
751:     /* Find CRL issuer: if not last certificate then issuer
752:      * is next certificate in chain.
753:      */
754:     if(cnum < chnum)
755:         issuer = sk_X509_value(ctx->chain, cnum + 1);
756:     else
757:     {
758:         issuer = sk_X509_value(ctx->chain, chnum);
759:         /* If not self signed, can't check signature */
760:         if(!ctx->check_issued(ctx, issuer, issuer))
761:         {
762:             ctx->error = X509_V_ERR_UNABLE_TO_GET_CRL_ISSUER;
763:             ok = ctx->verify_cb(0, ctx);
764:             if(!ok) goto err;
765:         }
766:     }
767:
768:     if(issuer)
769:     {
```

```
770:     /* Check for cRLSign bit if keyUsage present */
771:     if ((issuer->ex_flags & EXFLAG_KUSAGE) &&
772:         !(issuer->ex_kusage & KU_CRL_SIGN))
773:     {
774:         ctx->error = X509_V_ERR_KEYUSAGE_NO_CRL_SIGN;
775:         ok = ctx->verify_cb(0, ctx);
776:         if(!ok) goto err;
777:     }
778:
779:     /* Attempt to get issuer certificate public key */
780:     ikey = X509_get_pubkey(issuer);
781:
782:     if(!ikey)
783:     {
784:         ctx->error=X509_V_ERR_UNABLE_TO_DECODE_ISSUER_PUBLIC_KEY
785:     }
786:     else
787:     {
788:         ok = ctx->verify_cb(0, ctx);
789:         if (!ok) goto err;
790:     }
791:     /* Verify CRL signature */
792:     if(X509_CRL_verify(crl, ikey) <= 0)
793:     {
794:         ctx->error=X509_V_ERR_CRL_SIGNATURE_FAILURE;
795:         ok = ctx->verify_cb(0, ctx);
796:         if (!ok) goto err;
797:     }
798: }
799:
800: ok = check_crl_time(ctx, crl, 1);
801: if (!ok)
802:     goto err;
803:
804: ok = 1;
805:
806: err:
807: EVP_PKEY_free(ikey);
808: return ok;
809: }
810:
811: /* Check certificate against CRL */
812: static int cert_crl(X509_STORE_CTX *ctx, X509_CRL *crl, X509 *x)
813: {
814:     int idx, ok;
815:     X509_REVOKED rtmp;
816:     STACK_OF(X509_EXTENSION) *exts;
817:     X509_EXTENSION *ext;
818:     /* Look for serial number of certificate in CRL */
819:     rtmp.serialNumber = X509_get_serialNumber(x);
820:     /* Sort revoked into serial number order if not already sorted.
821:      * Do this under a lock to avoid race condition.
822:      */
823:     if (!sk_X509_REVOKED_is_sorted(crl->crl->revoked))
824:     {
825:         CRYPTO_w_lock(CRYPTO_LOCK_X509_CRL);
826:         sk_X509_REVOKED_sort(crl->crl->revoked);
827:         CRYPTO_w_unlock(CRYPTO_LOCK_X509_CRL);
828:     }
829:     idx = sk_X509_REVOKED_find(crl->crl->revoked, &rtmp);
830:     /* If found assume revoked: want something cleverer than
831:      * this to handle entry extensions in V2 CRLs.
832:      */
833:     if(idx >= 0)
```

```
834:         {
835:             ctx->error = X509_V_ERR_CERT_REVOKED;
836:             ok = ctx->verify_cb(0, ctx);
837:             if (!ok) return 0;
838:         }
839:
840:         if (ctx->param->flags & X509_V_FLAG_IGNORE_CRITICAL)
841:             return 1;
842:
843:         /* See if we have any critical CRL extensions: since we
844:          * currently don't handle any CRL extensions the CRL must be
845:          * rejected.
846:          * This code accesses the X509_CRL structure directly: applications
847:          * shouldn't do this.
848:         */
849:
850:         exts = crl->crl->extensions;
851:
852:         for (idx = 0; idx < sk_X509_EXTENSION_num(exts); idx++)
853:         {
854:             ext = sk_X509_EXTENSION_value(exts, idx);
855:             if (ext->critical > 0)
856:             {
857:                 ctx->error =
858:                     X509_V_ERR_UNHANDLED_CRITICAL_CRL_EXTENSION;
859:                 ok = ctx->verify_cb(0, ctx);
860:                 if(!ok) return 0;
861:                 break;
862:             }
863:         }
864:         return 1;
865:     }
866:
867: static int check_policy(X509_STORE_CTX *ctx)
868: {
869:     int ret;
870:     ret = X509_policy_check(&ctx->tree, &ctx->explicit_policy, ctx->chain,
871:                            ctx->param->policies, ctx->param->flags);
872:     if (ret == 0)
873:     {
874:         X509err(X509_F_CHECK_POLICY,ERR_R_MALLOC_FAILURE);
875:         return 0;
876:     }
877:     /* Invalid or inconsistent extensions */
878:     if (ret == -1)
879:     {
880:         /* Locate certificates with bad extensions and notify
881:          * callback.
882:          */
883:         X509 *x;
884:         int i;
885:         for (i = 1; i < sk_X509_num(ctx->chain); i++)
886:         {
887:             x = sk_X509_value(ctx->chain, i);
888:             if (!(x->ex_flags & EXFLAG_INVALID_POLICY))
889:                 continue;
890:             ctx->current_cert = x;
891:             ctx->error = X509_V_ERR_INVALID_POLICY_EXTENSION;
892:             ret = ctx->verify_cb(0, ctx);
893:         }
894:         return 1;
895:     }
896:     if (ret == -2)
897:     {
898:         ctx->current_cert = NULL;
```

```
899:         ctx->error = X509_V_ERR_NO_EXPLICIT_POLICY;
900:         return ctx->verify_cb(0, ctx);
901:     }
902:
903:     if (ctx->param->flags & X509_V_FLAG_NOTIFY_POLICY)
904:     {
905:         ctx->current_cert = NULL;
906:         ctx->error = X509_V_OK;
907:         if (!ctx->verify_cb(2, ctx))
908:             return 0;
909:     }
910:
911:     return 1;
912: }
913:
914: static int check_cert_time(X509_STORE_CTX *ctx, X509 *x)
915: {
916:     time_t *ptime;
917:     int i;
918:
919:     if (ctx->param->flags & X509_V_FLAG_USE_CHECK_TIME)
920:         ptime = &ctx->param->check_time;
921:     else
922:         ptime = NULL;
923:
924:     i=X509_cmp_time(X509_get_notBefore(x), ptime);
925:     if (i == 0)
926:     {
927:         ctx->error=X509_V_ERR_ERROR_IN_CERT_NOT_BEFORE_FIELD;
928:         ctx->current_cert=x;
929:         if (!ctx->verify_cb(0, ctx))
930:             return 0;
931:     }
932:
933:     if (i > 0)
934:     {
935:         ctx->error=X509_V_ERR_CERT_NOT_YET_VALID;
936:         ctx->current_cert=x;
937:         if (!ctx->verify_cb(0, ctx))
938:             return 0;
939:     }
940:
941:     i=X509_cmp_time(X509_get_notAfter(x), ptime);
942:     if (i == 0)
943:     {
944:         ctx->error=X509_V_ERR_ERROR_IN_CERT_NOT_AFTER_FIELD;
945:         ctx->current_cert=x;
946:         if (!ctx->verify_cb(0, ctx))
947:             return 0;
948:     }
949:
950:     if (i < 0)
951:     {
952:         ctx->error=X509_V_ERR_CERT_HAS_EXPIRED;
953:         ctx->current_cert=x;
954:         if (!ctx->verify_cb(0, ctx))
955:             return 0;
956:     }
957:
958:     return 1;
959: }
960:
961: static int internal_verify(X509_STORE_CTX *ctx)
962: {
963:     int ok=0,n;
```

```
964: X509 *xs,*xi;
965: EVP_PKEY *pkey=NULL;
966: int (*cb)(int xok,X509_STORE_CTX *xctx);
967:
968: cb=ctx->verify_cb;
969:
970: n=sk_X509_num(ctx->chain);
971: ctx->error_depth=n-1;
972: n--;
973: xi=sk_X509_value(ctx->chain,n);
974:
975: if (ctx->check_issued(ctx, xi, xi))
976:     xs=xi;
977: else
978: {
979:     if (n <= 0)
980:     {
981:         ctx->error=X509_V_ERR_UNABLE_TO_VERIFY_LEAF_SIGNATURE;
982:         ctx->current_cert=xi;
983:         ok=cb(0,ctx);
984:         goto end;
985:     }
986:     else
987:     {
988:         n--;
989:         ctx->error_depth=n;
990:         xs=sk_X509_value(ctx->chain,n);
991:     }
992: }
993:
994: /*      ctx->error=0;  not needed */
995: while (n >= 0)
996: {
997:     ctx->error_depth=n;
998:     if (!xs->valid)
999:     {
1000:         if ((pkey=X509_get_pubkey(xi)) == NULL)
1001:         {
1002:             ctx->current_cert=xi;
1003:             ok=(*cb)(0,ctx);
1004:             if (!ok) goto end;
1005:         }
1006:         else if (X509_verify(xs,pkey) <= 0)
1007:             /* XXX For the final trusted self-signed cert,
1008:                * this is a waste of time. That check should
1009:                * optional so that e.g. 'openssl x509' can be
1010:                * used to detect invalid self-signatures, but
1011:                * we don't verify again and again in SSL
1012:                * handshakes and the like once the cert has
1013:                * been declared trusted. */
1014:         {
1015:             ctx->error=X509_V_ERR_CERT_SIGNATURE_FAILURE;
1016:             ctx->current_cert=xs;
1017:             ok=(*cb)(0,ctx);
1018:             if (!ok)
1019:             {
1020:                 EVP_PKEY_free(pkey);
1021:                 goto end;
1022:             }
1023:         }
1024:     }
1025:     EVP_PKEY_free(pkey);
1026:     pkey=NULL;
1027: }
```

```
1028:           xs->valid = 1;
1029:
1030:
1031:           ok = check_cert_time(ctx, xs);
1032:           if (!ok)
1033:               goto end;
1034:
1035: /* The last error (if any) is still in the error value */
1036:           ctx->current_issuer=xi;
1037:           ctx->current_cert=xs;
1038:           ok=(*cb)(1,ctx);
1039:           if (!ok) goto end;
1040:
1041:           n--;
1042:           if (n >= 0)
1043:           {
1044:               xi=xs;
1045:               xs=sk_X509_value(ctx->chain,n);
1046:           }
1047:       }
1048:   ok=1;
1049: end:
1050:     return ok;
1051: }
1052:
1053: int X509_cmp_current_time(ASN1_TIME *ctm)
1054: {
1055:     return X509_cmp_time(ctm, NULL);
1056: }
1057:
1058: int X509_cmp_time(ASN1_TIME *ctm, time_t *cmp_time)
1059: {
1060:     char *str;
1061:     ASN1_TIME atm;
1062:     long offset;
1063:     char buff1[24],buff2[24],*p;
1064:     int i,j;
1065:
1066:     p=buff1;
1067:     i=ctm->length;
1068:     str=(char *)ctm->data;
1069:     if (ctm->type == V_ASN1_UTCTIME)
1070:     {
1071:         if ((i < 11) || (i > 17)) return 0;
1072:         memcpy(p,str,10);
1073:         p+=10;
1074:         str+=10;
1075:     }
1076: else
1077: {
1078:     if (i < 13) return 0;
1079:     memcpy(p,str,12);
1080:     p+=12;
1081:     str+=12;
1082: }
1083:
1084: if ((*str == 'Z') || (*str == '-') || (*str == '+'))
1085: { *(p++)='0'; *(p++)='0'; }
1086: else
1087: {
1088:     *(p++)= *(str++);
1089:     *(p++)= *(str++);
1090:     /* Skip any fractional seconds... */
1091:     if (*str == '.')
1092:     {
```

```
1093:                     str++;
1094:                     while ((*str >= '0') && (*str <= '9')) str++;
1095:                 }
1096:             }
1097:         }
1098:         *(p++)='Z';
1099:         *(p++)='\0';
1100:
1101:         if (*str == 'Z')
1102:             offset=0;
1103:         else
1104:             {
1105:                 if ((*str != '+') && (*str != '-'))
1106:                     return 0;
1107:                 offset=((str[1]-'0')*10+(str[2]-'0'))*60;
1108:                 offset+=(str[3]-'0')*10+(str[4]-'0');
1109:                 if (*str == '-')
1110:                     offset= -offset;
1111:             }
1112:         atm.type=ctm->type;
1113:         atm.length=sizeof(buff2);
1114:         atm.data=(unsigned char *)buff2;
1115:
1116:         if (X509_time_adj(&atm,-offset*60, cmp_time) == NULL)
1117:             return 0;
1118:
1119:         if (ctm->type == V ASN1_UTCTIME)
1120:             {
1121:                 i=(buff1[0]-'0')*10+(buff1[1]-'0');
1122:                 if (i < 50) i+=100; /* cf. RFC 2459 */
1123:                 j=(buff2[0]-'0')*10+(buff2[1]-'0');
1124:                 if (j < 50) j+=100;
1125:
1126:                 if (i < j) return -1;
1127:                 if (i > j) return 1;
1128:             }
1129:         i=strcmp(buff1,buff2);
1130:         if (i == 0) /* wait a second then return younger :-) */
1131:             return -1;
1132:         else
1133:             return i;
1134:     }
1135:
1136: ASN1_TIME *X509_gmtime_adj(ASN1_TIME *s, long adj)
1137: {
1138:     return X509_time_adj(s, adj, NULL);
1139: }
1140:
1141: ASN1_TIME *X509_time_adj(ASN1_TIME *s, long adj, time_t *in_tm)
1142: {
1143:     time_t t;
1144:     int type = -1;
1145:
1146:     if (in_tm) t = *in_tm;
1147:     else time(&t);
1148:
1149:     t+=adj;
1150:     if (s) type = s->type;
1151:     if (type == V ASN1_UTCTIME) return ASN1_UTCTIME_set(s,t);
1152:     if (type == V ASN1_GENERALIZEDTIME) return ASN1_GENERALIZEDTIME_set(s, t
);
1153:     return ASN1_TIME_set(s, t);
1154: }
1155:
1156: int X509_get_pubkey_parameters(EVP_PKEY *pkey, STACK_OF(X509) *chain)
```

```

x509_vfy.c      Mon Apr 30 19:22:19 2007      19

1157: {
1158:     EVP_PKEY *ktmp=NULL, *ktmp2;
1159:     int i,j;
1160:
1161:     if ((pkey != NULL) && !EVP_PKEY_missing_parameters(pkey)) return 1;
1162:
1163:     for (i=0; i<sk_X509_num(chain); i++)
1164:     {
1165:         ktmp=X509_get_pubkey(sk_X509_value(chain,i));
1166:         if (ktmp == NULL)
1167:         {
1168:             X509err(X509_F_X509_GET_PUBKEY_PARAMETERS,X509_R_UNABLE_
TO_GET_CERTS_PUBLIC_KEY);
1169:             return 0;
1170:         }
1171:         if (!EVP_PKEY_missing_parameters(ktmp))
1172:             break;
1173:         else
1174:         {
1175:             EVP_PKEY_free(ktmp);
1176:             ktmp=NULL;
1177:         }
1178:     }
1179:     if (ktmp == NULL)
1180:     {
1181:         X509err(X509_F_X509_GET_PUBKEY_PARAMETERS,X509_R_UNABLE_TO_FIND_
PARAMETERS_IN_CHAIN);
1182:         return 0;
1183:     }
1184:
1185:     /* first, populate the other certs */
1186:     for (j=i-1; j >= 0; j--)
1187:     {
1188:         ktmp2=X509_get_pubkey(sk_X509_value(chain,j));
1189:         EVP_PKEY_copy_parameters(ktmp2,ktmp);
1190:         EVP_PKEY_free(ktmp2);
1191:     }
1192:
1193:     if (pkey != NULL) EVP_PKEY_copy_parameters(pkey,ktmp);
1194:     EVP_PKEY_free(ktmp);
1195:     return 1;
1196: }
1197:
1198: int X509_STORE_CTX_get_ex_new_index(long argl, void *argp, CRYPTO_EX_new *new_fu
nc,
1199:                                     CRYPTO_EX_dup *dup_func, CRYPTO_EX_free *free_func)
1200: {
1201:     /* This function is (usually) called only once, by
1202:      * SSL_get_ex_data_X509_STORE_CTX_idx (ssl/ssl_cert.c). */
1203:     return CRYPTO_get_ex_new_index(CRYPTO_EX_INDEX_X509_STORE_CTX, argl, arg
p,
1204:                                     new_func, dup_func, free_func);
1205: }
1206:
1207: int X509_STORE_CTX_set_ex_data(X509_STORE_CTX *ctx, int idx, void *data)
1208: {
1209:     return CRYPTO_set_ex_data(&ctx->ex_data,idx,data);
1210: }
1211:
1212: void *X509_STORE_CTX_get_ex_data(X509_STORE_CTX *ctx, int idx)
1213: {
1214:     return CRYPTO_get_ex_data(&ctx->ex_data,idx);
1215: }
1216:
1217: int X509_STORE_CTX_get_error(X509_STORE_CTX *ctx)

```

```
1218:     {
1219:         return ctx->error;
1220:     }
1221:
1222: void X509_STORE_CTX_set_error(X509_STORE_CTX *ctx, int err)
1223: {
1224:     ctx->error=err;
1225: }
1226:
1227: int X509_STORE_CTX_get_error_depth(X509_STORE_CTX *ctx)
1228: {
1229:     return ctx->error_depth;
1230: }
1231:
1232: X509 *X509_STORE_CTX_get_current_cert(X509_STORE_CTX *ctx)
1233: {
1234:     return ctx->current_cert;
1235: }
1236:
1237: STACK_OF(X509) *X509_STORE_CTX_get_chain(X509_STORE_CTX *ctx)
1238: {
1239:     return ctx->chain;
1240: }
1241:
1242: STACK_OF(X509) *X509_STORE_CTX_get1_chain(X509_STORE_CTX *ctx)
1243: {
1244:     int i;
1245:     X509 *x;
1246:     STACK_OF(X509) *chain;
1247:     if (!ctx->chain || !(chain = sk_X509_dup(ctx->chain))) return NULL;
1248:     for (i = 0; i < sk_X509_num(chain); i++)
1249:     {
1250:         x = sk_X509_value(chain, i);
1251:         CRYPTO_add(&x->references, 1, CRYPTO_LOCK_X509);
1252:     }
1253:     return chain;
1254: }
1255:
1256: void X509_STORE_CTX_set_cert(X509_STORE_CTX *ctx, X509 *x)
1257: {
1258:     ctx->cert=x;
1259: }
1260:
1261: void X509_STORE_CTX_set_chain(X509_STORE_CTX *ctx, STACK_OF(X509) *sk)
1262: {
1263:     ctx->untrusted=sk;
1264: }
1265:
1266: void X509_STORE_CTX_set0_crls(X509_STORE_CTX *ctx, STACK_OF(X509_CRL) *sk)
1267: {
1268:     ctx->crls=sk;
1269: }
1270:
1271: int X509_STORE_CTX_set_purpose(X509_STORE_CTX *ctx, int purpose)
1272: {
1273:     return X509_STORE_CTX_purpose_inherit(ctx, 0, purpose, 0);
1274: }
1275:
1276: int X509_STORE_CTX_set_trust(X509_STORE_CTX *ctx, int trust)
1277: {
1278:     return X509_STORE_CTX_purpose_inherit(ctx, 0, 0, trust);
1279: }
1280:
1281: /* This function is used to set the X509_STORE_CTX purpose and trust
1282:  * values. This is intended to be used when another structure has its
```

```
1283: * own trust and purpose values which (if set) will be inherited by
1284: * the ctx. If they aren't set then we will usually have a default
1285: * purpose in mind which should then be used to set the trust value.
1286: * An example of this is SSL use: an SSL structure will have its own
1287: * purpose and trust settings which the application can set: if they
1288: * aren't set then we use the default of SSL client/server.
1289: */
1290:
1291: int X509_STORE_CTX_purpose_inherit(X509_STORE_CTX *ctx, int def_purpose,
1292:                                     int purpose, int trust)
1293: {
1294:     int idx;
1295:     /* If purpose not set use default */
1296:     if (!purpose) purpose = def_purpose;
1297:     /* If we have a purpose then check it is valid */
1298:     if (purpose)
1299:     {
1300:         X509_PURPOSE *ptmp;
1301:         idx = X509_PURPOSE_get_by_id(purpose);
1302:         if (idx == -1)
1303:         {
1304:             X509err(X509_F_X509_STORE_CTX_PURPOSE_INHERIT,
1305:                     X509_R_UNKNOWN_PURPOSE_ID);
1306:             return 0;
1307:         }
1308:         ptmp = X509_PURPOSE_get0(idx);
1309:         if (ptmp->trust == X509_TRUST_DEFAULT)
1310:         {
1311:             idx = X509_PURPOSE_get_by_id(def_purpose);
1312:             if (idx == -1)
1313:             {
1314:                 X509err(X509_F_X509_STORE_CTX_PURPOSE_INHERIT,
1315:                         X509_R_UNKNOWN_PURPOSE_ID);
1316:                 return 0;
1317:             }
1318:             ptmp = X509_PURPOSE_get0(idx);
1319:         }
1320:         /* If trust not set then get from purpose default */
1321:         if (!trust) trust = ptmp->trust;
1322:     }
1323:     if (trust)
1324:     {
1325:         idx = X509_TRUST_get_by_id(trust);
1326:         if (idx == -1)
1327:         {
1328:             X509err(X509_F_X509_STORE_CTX_PURPOSE_INHERIT,
1329:                     X509_R_UNKNOWN_TRUST_ID);
1330:             return 0;
1331:         }
1332:     }
1333:
1334:     if (purpose && !ctx->param->purpose) ctx->param->purpose = purpose;
1335:     if (trust && !ctx->param->trust) ctx->param->trust = trust;
1336:     return 1;
1337: }
1338:
1339: X509_STORE_CTX *X509_STORE_CTX_new(void)
1340: {
1341:     X509_STORE_CTX *ctx;
1342:     ctx = (X509_STORE_CTX *)OPENSSL_malloc(sizeof(X509_STORE_CTX));
1343:     if (!ctx)
1344:     {
1345:         X509err(X509_F_X509_STORE_CTX_NEW,ERR_R_MALLOC_FAILURE);
1346:         return NULL;
1347:     }
```

```
x509_vfy.c      Mon Apr 30 19:22:19 2007      22
1348:     memset(ctx, 0, sizeof(X509_STORE_CTX));
1349:     return ctx;
1350: }
1351:
1352: void X509_STORE_CTX_free(X509_STORE_CTX *ctx)
1353: {
1354:     X509_STORE_CTX_cleanup(ctx);
1355:     OPENSSL_free(ctx);
1356: }
1357:
1358: int X509_STORE_CTX_init(X509_STORE_CTX *ctx, X509_STORE *store, X509 *x509,
1359:                         STACK_OF(X509) *chain)
1360: {
1361:     int ret = 1;
1362:     ctx->ctx=store;
1363:     ctx->current_method=0;
1364:     ctx->cert=x509;
1365:     ctx->untrusted=chain;
1366:     ctx->crls = NULL;
1367:     ctx->last_untrusted=0;
1368:     ctx->other_ctx=NULL;
1369:     ctx->valid=0;
1370:     ctx->chain=NULL;
1371:     ctx->error=0;
1372:     ctx->explicit_policy=0;
1373:     ctx->error_depth=0;
1374:     ctx->current_cert=NULL;
1375:     ctx->current_issuer=NULL;
1376:     ctx->tree = NULL;
1377:
1378:     ctx->param = X509_VERIFY_PARAM_new();
1379:
1380:     if (!ctx->param)
1381:     {
1382:         X509err(X509_F_X509_STORE_CTX_INIT,ERR_R_MALLOC_FAILURE);
1383:         return 0;
1384:     }
1385:
1386: /* Inherit callbacks and flags from X509_STORE if not set
1387:  * use defaults.
1388: */
1389:
1390:
1391:     if (store)
1392:         ret = X509_VERIFY_PARAM_inherit(ctx->param, store->param);
1393:     else
1394:         ctx->param->flags |= X509_VP_FLAG_DEFAULT|X509_VP_FLAG_ONCE;
1395:
1396:     if (store)
1397:     {
1398:         ctx->verify_cb = store->verify_cb;
1399:         ctx->cleanup = store->cleanup;
1400:     }
1401:     else
1402:         ctx->cleanup = 0;
1403:
1404:     if (ret)
1405:         ret = X509_VERIFY_PARAM_inherit(ctx->param,
1406:                                         X509_VERIFY_PARAM_lookup("default"));
1407:
1408:     if (ret == 0)
1409:     {
1410:         X509err(X509_F_X509_STORE_CTX_INIT,ERR_R_MALLOC_FAILURE);
1411:         return 0;
1412:     }
```

```
1413:  
1414:     if (store && store->check_issued)  
1415:         ctx->check_issued = store->check_issued;  
1416:     else  
1417:         ctx->check_issued = check_issued;  
1418:  
1419:     if (store && store->get_issuer)  
1420:         ctx->get_issuer = store->get_issuer;  
1421:     else  
1422:         ctx->get_issuer = X509_STORE_CTX_get1_issuer;  
1423:  
1424:     if (store && store->verify_cb)  
1425:         ctx->verify_cb = store->verify_cb;  
1426:     else  
1427:         ctx->verify_cb = null_callback;  
1428:  
1429:     if (store && store->verify)  
1430:         ctx->verify = store->verify;  
1431:     else  
1432:         ctx->verify = internal_verify;  
1433:  
1434:     if (store && store->check_revocation)  
1435:         ctx->check_revocation = store->check_revocation;  
1436:     else  
1437:         ctx->check_revocation = check_revocation;  
1438:  
1439:     if (store && store->get_crl)  
1440:         ctx->get_crl = store->get_crl;  
1441:     else  
1442:         ctx->get_crl = get_crl;  
1443:  
1444:     if (store && store->check_crl)  
1445:         ctx->check_crl = store->check_crl;  
1446:     else  
1447:         ctx->check_crl = check_crl;  
1448:  
1449:     if (store && store->cert_crl)  
1450:         ctx->cert_crl = store->cert_crl;  
1451:     else  
1452:         ctx->cert_crl = cert_crl;  
1453:  
1454:     ctx->check_policy = check_policy;  
1455:  
1456:  
1457: /* This memset() can't make any sense anyway, so it's removed. As  
1458: * X509_STORE_CTX_cleanup does a proper "free" on the ex_data, we put a  
1459: * corresponding "new" here and remove this bogus initialisation. */  
1460: /* memset(&(ctx->ex_data),0,sizeof(CRYPTO_EX_DATA)); */  
1461: if (!CRYPTO_new_ex_data(CRYPTO_EX_INDEX_X509_STORE_CTX, ctx,  
1462:                         &(ctx->ex_data)))  
1463: {  
1464:     OPENSSL_free(ctx);  
1465:     X509err(X509_F_X509_STORE_CTX_INIT,ERR_R_MALLOC_FAILURE);  
1466:     return 0;  
1467: }  
1468: return 1;  
1469: }  
1470:  
1471: /* Set alternative lookup method: just a STACK of trusted certificates.  
1472: * This avoids X509_STORE nastiness where it isn't needed.  
1473: */  
1474:  
1475: void X509_STORE_CTX_trusted_stack(X509_STORE_CTX *ctx, STACK_OF(X509) *sk)  
1476: {  
1477:     ctx->other_ctx = sk;
```

```

x509_vfy.c      Mon Apr 30 19:22:19 2007      24
1478:           ctx->get_issuer = get_issuer_sk;
1479:     }
1480:
1481: void X509_STORE_CTX_cleanup(X509_STORE_CTX *ctx)
1482: {
1483:   if (ctx->cleanup) ctx->cleanup(ctx);
1484:   if (ctx->param != NULL)
1485:   {
1486:     X509_VERIFY_PARAM_free(ctx->param);
1487:     ctx->param=NULL;
1488:   }
1489:   if (ctx->tree != NULL)
1490:   {
1491:     X509_policy_tree_free(ctx->tree);
1492:     ctx->tree=NULL;
1493:   }
1494:   if (ctx->chain != NULL)
1495:   {
1496:     sk_X509_pop_free(ctx->chain,X509_free);
1497:     ctx->chain=NULL;
1498:   }
1499: CRYPTO_free_ex_data(CRYPTO_EX_INDEX_X509_STORE_CTX, ctx, &(ctx->ex_data))
);
1500: memset(&ctx->ex_data,0,sizeof(CRYPTO_EX_DATA));
1501: }
1502:
1503: void X509_STORE_CTX_set_depth(X509_STORE_CTX *ctx, int depth)
1504: {
1505:   X509_VERIFY_PARAM_set_depth(ctx->param, depth);
1506: }
1507:
1508: void X509_STORE_CTX_set_flags(X509_STORE_CTX *ctx, unsigned long flags)
1509: {
1510:   X509_VERIFY_PARAM_set_flags(ctx->param, flags);
1511: }
1512:
1513: void X509_STORE_CTX_set_time(X509_STORE_CTX *ctx, unsigned long flags, time_t t)
1514: {
1515:   X509_VERIFY_PARAM_set_time(ctx->param, t);
1516: }
1517:
1518: void X509_STORE_CTX_set_verify_cb(X509_STORE_CTX *ctx,
1519:                                     int (*verify_cb)(int, X509_STORE_CTX *));
1520: {
1521:   ctx->verify_cb=verify_cb;
1522: }
1523:
1524: X509_POLICY_TREE *X509_STORE_CTX_get0_policy_tree(X509_STORE_CTX *ctx)
1525: {
1526:   return ctx->tree;
1527: }
1528:
1529: int X509_STORE_CTX_get_explicit_policy(X509_STORE_CTX *ctx)
1530: {
1531:   return ctx->explicit_policy;
1532: }
1533:
1534: int X509_STORE_CTX_set_default(X509_STORE_CTX *ctx, const char *name)
1535: {
1536:   const X509_VERIFY_PARAM *param;
1537:   param = X509_VERIFY_PARAM_lookup(name);
1538:   if (!param)
1539:     return 0;
1540:   return X509_VERIFY_PARAM_inherit(ctx->param, param);
1541: }

```

```
1542:  
1543: X509_VERIFY_PARAM *X509_STORE_CTX_get0_param(X509_STORE_CTX *ctx)  
1544: {  
1545:     return ctx->param;  
1546: }  
1547:  
1548: void X509_STORE_CTX_set0_param(X509_STORE_CTX *ctx, X509_VERIFY_PARAM *param)  
1549: {  
1550:     if (ctx->param)  
1551:         X509_VERIFY_PARAM_free(ctx->param);  
1552:     ctx->param = param;  
1553: }  
1554:  
1555: IMPLEMENT_STACK_OF(X509)  
1556: IMPLEMENT ASN1_SET_OF(X509)  
1557:  
1558: IMPLEMENT_STACK_OF(X509_NAME)  
1559:  
1560: IMPLEMENT_STACK_OF(X509_ATTRIBUTE)  
1561: IMPLEMENT ASN1_SET_OF(X509_ATTRIBUTE)
```

Attachment 5

v3_purp.c

```
1: /* v3_purp.c */
2: /* Written by Dr Stephen N Henson (shenson@bigfoot.com) for the OpenSSL
3: * project 2001.
4: */
5: /**
6: * Copyright (c) 1999-2004 The OpenSSL Project. All rights reserved.
7: *
8: * Redistribution and use in source and binary forms, with or without
9: * modification, are permitted provided that the following conditions
10: * are met:
11: *
12: * 1. Redistributions of source code must retain the above copyright
13: * notice, this list of conditions and the following disclaimer.
14: *
15: * 2. Redistributions in binary form must reproduce the above copyright
16: * notice, this list of conditions and the following disclaimer in
17: * the documentation and/or other materials provided with the
18: * distribution.
19: *
20: * 3. All advertising materials mentioning features or use of this
21: * software must display the following acknowledgment:
22: * "This product includes software developed by the OpenSSL Project
23: * for use in the OpenSSL Toolkit. (http://www.OpenSSL.org/)"
24: *
25: * 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
26: * endorse or promote products derived from this software without
27: * prior written permission. For written permission, please contact
28: * licensing@OpenSSL.org.
29: *
30: * 5. Products derived from this software may not be called "OpenSSL"
31: * nor may "OpenSSL" appear in their names without prior written
32: * permission of the OpenSSL Project.
33: *
34: * 6. Redistributions of any form whatsoever must retain the following
35: * acknowledgment:
36: * "This product includes software developed by the OpenSSL Project
37: * for use in the OpenSSL Toolkit (http://www.OpenSSL.org/)"
38: *
39: * THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ''AS IS'' AND ANY
40: * EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
41: * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
42: * PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE OpenSSL PROJECT OR
43: * ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
44: * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
45: * NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
46: * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
47: * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
48: * STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
49: * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
50: * OF THE POSSIBILITY OF SUCH DAMAGE.
51: */
52: *
53: * This product includes cryptographic software written by Eric Young
54: * (eay@cryptsoft.com). This product includes software written by Tim
55: * Hudson (tjh@cryptsoft.com).
56: *
57: */
58:
59: #include <stdio.h>
60: #include "cryptlib.h"
61: #include "x509/globus_proxycert.h"
62: #include <openssl/x509v3.h>
63: #include <openssl/x509_vfy.h>
64:
65: static void x509v3_cache_extensions(X509 *x);
```

```
66:
67: static int check_ssl_ca(const X509 *x);
68: static int check_purpose_ssl_client(const X509_PURPOSE *xp, const X509 *x, int ca);
69: static int check_purpose_ssl_server(const X509_PURPOSE *xp, const X509 *x, int ca);
70: static int check_purpose_ns_ssl_server(const X509_PURPOSE *xp, const X509 *x, int ca);
71: static int purpose_smime(const X509 *x, int ca);
72: static int check_purpose_smime_sign(const X509_PURPOSE *xp, const X509 *x, int ca);
73: static int check_purpose_smime_encrypt(const X509_PURPOSE *xp, const X509 *x, int ca);
74: static int check_purpose_crl_sign(const X509_PURPOSE *xp, const X509 *x, int ca);
75: static int no_check(const X509_PURPOSE *xp, const X509 *x, int ca);
76: static int ocsp_helper(const X509_PURPOSE *xp, const X509 *x, int ca);
77:
78: static int xp_cmp(const X509_PURPOSE * const *a,
79:                   const X509_PURPOSE * const *b);
80: static void xptable_free(X509_PURPOSE *p);
81:
82: static X509_PURPOSE xstandard[] = {
83:     {X509_PURPOSE_SSL_CLIENT, X509_TRUST_SSL_CLIENT, 0, check_purpose_ssl_client, "SSL client", "sslclient", NULL},
84:     {X509_PURPOSE_SSL_SERVER, X509_TRUST_SSL_SERVER, 0, check_purpose_ssl_server, "SSL server", "sslserver", NULL},
85:     {X509_PURPOSE_NS_SSL_SERVER, X509_TRUST_SSL_SERVER, 0, check_purpose_ns_ssl_server, "Netscape SSL server", "nssslserver", NULL},
86:     {X509_PURPOSE_SMIME_SIGN, X509_TRUST_EMAIL, 0, check_purpose_smime_sign, "S/MIME signing", "smimesign", NULL},
87:     {X509_PURPOSE_SMIME_ENCRYPT, X509_TRUST_EMAIL, 0, check_purpose_smime_encrypt, "S/MIME encryption", "smimeencrypt", NULL},
88:     {X509_PURPOSE_CRL_SIGN, X509_TRUST_COMPAT, 0, check_purpose_crl_sign, "CRL signing", "crlsign", NULL},
89:     {X509_PURPOSE_ANY, X509_TRUST_DEFAULT, 0, no_check, "Any Purpose", "any", NULL},
90:     {X509_PURPOSE_OCSP_HELPER, X509_TRUST_COMPAT, 0, ocsp_helper, "OCSP helper", "ocspHelper", NULL},
91: };
92:
93: #define X509_PURPOSE_COUNT (sizeof(xstandard)/sizeof(X509_PURPOSE))
94:
95: IMPLEMENT_STACK_OF(X509_PURPOSE)
96:
97: static STACK_OF(X509_PURPOSE) *xptable = NULL;
98:
99: static int xp_cmp(const X509_PURPOSE * const *a,
100:                   const X509_PURPOSE * const *b)
101: {
102:     return (*a)->purpose - (*b)->purpose;
103: }
104:
105: /* As much as I'd like to make X509_check_purpose use a "const" X509*
106:  * I really can't because it does recalculate hashes and do other non-const
107:  * things. */
108: int X509_check_purpose(X509 *x, int id, int ca)
109: {
110:     int idx;
111:     const X509_PURPOSE *pt;
112:     if(!(x->ex_flags & EXFLAG_SET)) {
113:         CRYPTO_w_lock(CRYPTO_LOCK_X509);
114:         x509v3_cache_extensions(x);
115:         CRYPTO_w_unlock(CRYPTO_LOCK_X509);
116:     }
}
```

```

v3_purp.c      Mon Apr 30 19:22:42 2007      3

117:     if(id == -1) return 1;
118:     idx = X509_PURPOSE_get_by_id(id);
119:     if(idx == -1) return -1;
120:     pt = X509_PURPOSE_get0(idx);
121:     return pt->check_purpose(pt, x, ca);
122: }
123:
124: int X509_PURPOSE_set(int *p, int purpose)
125: {
126:     if(X509_PURPOSE_get_by_id(purpose) == -1) {
127:         X509V3err(X509V3_F_X509_PURPOSE_SET, X509V3_R_INVALID_PURPOSE);
128:         return 0;
129:     }
130:     *p = purpose;
131:     return 1;
132: }
133:
134: int X509_PURPOSE_get_count(void)
135: {
136:     if(!xptable) return X509_PURPOSE_COUNT;
137:     return sk_X509_PURPOSE_num(xptable) + X509_PURPOSE_COUNT;
138: }
139:
140: X509_PURPOSE * X509_PURPOSE_get0(int idx)
141: {
142:     if(idx < 0) return NULL;
143:     if(idx < (int)X509_PURPOSE_COUNT) return xstandard + idx;
144:     return sk_X509_PURPOSE_value(xptable, idx - X509_PURPOSE_COUNT);
145: }
146:
147: int X509_PURPOSE_get_by_sname(char *sname)
148: {
149:     int i;
150:     X509_PURPOSE *xptmp;
151:     for(i = 0; i < X509_PURPOSE_get_count(); i++) {
152:         xptmp = X509_PURPOSE_get0(i);
153:         if(!strcmp(xptmp->sname, sname)) return i;
154:     }
155:     return -1;
156: }
157:
158: int X509_PURPOSE_get_by_id(int purpose)
159: {
160:     X509_PURPOSE tmp;
161:     int idx;
162:     if((purpose >= X509_PURPOSE_MIN) && (purpose <= X509_PURPOSE_MAX))
163:         return purpose - X509_PURPOSE_MIN;
164:     tmp.purpose = purpose;
165:     if(!xptable) return -1;
166:     idx = sk_X509_PURPOSE_find(xptable, &tmp);
167:     if(idx == -1) return -1;
168:     return idx + X509_PURPOSE_COUNT;
169: }
170:
171: int X509_PURPOSE_add(int id, int trust, int flags,
172:                       int (*ck)(const X509_PURPOSE *, const X509 *, int),
173:                       char *name, char *sname, void *arg)
174: {
175:     int idx;
176:     X509_PURPOSE *ptmp;
177:     /* This is set according to what we change: application can't set it */
178:     flags &= ~X509_PURPOSE_DYNAMIC;
179:     /* This will always be set for application modified trust entries */
180:     flags |= X509_PURPOSE_DYNAMIC_NAME;
181:     /* Get existing entry if any */

```

```

v3_purp.c      Mon Apr 30 19:22:42 2007      4
182:     idx = X509_PURPOSE_get_by_id(id);
183:     /* Need a new entry */
184:     if(idx == -1) {
185:         if(!(ptmp = OPENSSL_malloc(sizeof(X509_PURPOSE)))) {
186:             X509V3err(X509V3_F_X509_PURPOSE_ADD,ERR_R_MALLOC_FAILURE
187:         );
188:             return 0;
189:         }
190:         ptmp->flags = X509_PURPOSE_DYNAMIC;
191:     } else ptmp = X509_PURPOSE_get0(idx);
192:     /* OPENSSL_free existing name if dynamic */
193:     if(ptmp->flags & X509_PURPOSE_DYNAMIC_NAME) {
194:         OPENSSL_free(ptmp->name);
195:         OPENSSL_free(ptmp->sname);
196:     }
197:     /* dup supplied name */
198:     ptmp->name = BUF_strdup(name);
199:     ptmp->sname = BUF_strdup(sname);
200:     if(!ptmp->name || !ptmp->sname) {
201:         X509V3err(X509V3_F_X509_PURPOSE_ADD,ERR_R_MALLOC_FAILURE);
202:         return 0;
203:     }
204:     /* Keep the dynamic flag of existing entry */
205:     ptmp->flags &= X509_PURPOSE_DYNAMIC;
206:     /* Set all other flags */
207:     ptmp->flags |= flags;
208:
209:     ptmp->purpose = id;
210:     ptmp->trust = trust;
211:     ptmp->check_purpose = ck;
212:     ptmp->usr_data = arg;
213:
214:     /* If its a new entry manage the dynamic table */
215:     if(idx == -1) {
216:         if(!xptable && !(xptable = sk_X509_PURPOSE_new(xp_cmp))) {
217:             X509V3err(X509V3_F_X509_PURPOSE_ADD,ERR_R_MALLOC_FAILURE
218:         );
219:             return 0;
220:         }
221:         if (!sk_X509_PURPOSE_push(xptable, ptmp)) {
222:             X509V3err(X509V3_F_X509_PURPOSE_ADD,ERR_R_MALLOC_FAILURE
223:         );
224:         }
225:         return 1;
226:     }
227:
228: static void xptable_free(X509_PURPOSE *p)
229: {
230:     if(!p) return;
231:     if (p->flags & X509_PURPOSE_DYNAMIC)
232:     {
233:         if (p->flags & X509_PURPOSE_DYNAMIC_NAME) {
234:             OPENSSL_free(p->name);
235:             OPENSSL_free(p->sname);
236:         }
237:         OPENSSL_free(p);
238:     }
239: }
240:
241: void X509_PURPOSE_cleanup(void)
242: {
243:     unsigned int i;

```

```

v3_purp.c      Mon Apr 30 19:22:42 2007      5
244:     sk_X509_PURPOSE_pop_free(xptable, xptable_free);
245:     for(i = 0; i < X509_PURPOSE_COUNT; i++) xptable_free(xstandard + i);
246:     xptable = NULL;
247: }
248:
249: int X509_PURPOSE_get_id(X509_PURPOSE *xp)
250: {
251:     return xp->purpose;
252: }
253:
254: char *X509_PURPOSE_get0_name(X509_PURPOSE *xp)
255: {
256:     return xp->name;
257: }
258:
259: char *X509_PURPOSE_get0_sname(X509_PURPOSE *xp)
260: {
261:     return xp->sname;
262: }
263:
264: int X509_PURPOSE_get_trust(X509_PURPOSE *xp)
265: {
266:     return xp->trust;
267: }
268:
269: static int nid_cmp(int *a, int *b)
270: {
271:     return *a - *b;
272: }
273:
274: int X509_supported_extension(X509_EXTENSION *ex)
275: {
276:     /* This table is a list of the NIDs of supported extensions:
277:      * that is those which are used by the verify process. If
278:      * an extension is critical and doesn't appear in this list
279:      * then the verify process will normally reject the certificate.
280:      * The list must be kept in numerical order because it will be
281:      * searched using bsearch.
282:     */
283:
284:     static int supported_nids[] = {
285:         NID_netscape_cert_type, /* 71 */
286:         NID_key_usage, /* 83 */
287:         NID_subject_alt_name, /* 85 */
288:         NID_basic_constraints, /* 87 */
289:         NID_certificate_policies, /* 89 */
290:         NID_ext_key_usage, /* 126 */
291: #ifndef OPENSSL_NO_RFC3779
292:         NID_sbgp_ipAddrBlock, /* 290 */
293:         NID_sbgp_autonomousSysNum, /* 291 */
294: #endif
295:         NID_proxyCertInfo, /* 663 */
296:         NID_globusProxyCertInfo /* 772 */
297:     };
298:
299:     int ex_nid;
300:
301:     ex_nid = OBJ_obj2nid(X509_EXTENSION_get_object(ex));
302:
303:     if (ex_nid == NID_undef)
304:         return 0;
305:
306:     if (OBJ_bsearch((char *)&ex_nid, (char *)supported_nids,
307:                     sizeof(supported_nids)/sizeof(int), sizeof(int),
308:                     (int (*) (const void *, const void *))nid_cmp))

```

```
309:         return 1;
310:     return 0;
311: }
312:
313:
314: static void x509v3_cache_extensions(X509 *x)
315: {
316:     BASIC_CONSTRAINTS *bs;
317:     PROXY_CERT_INFO_EXTENSION *pci;
318:     PROXYCERTINFO *pci_old;
319:     ASN1_BIT_STRING *usage;
320:     ASN1_BIT_STRING *ns;
321:     EXTENDED_KEY_USAGE *extusage;
322:     X509_EXTENSION *ex;
323:
324:     int i;
325:     if(x->ex_flags & EXFLAG_SET) return;
326: #ifndef OPENSSL_NO_SHA
327:     X509_digest(x, EVP_sha1(), x->sha1_hash, NULL);
328: #endif
329:     /* Does subject name match issuer ? */
330:     if(!X509_NAME_cmp(X509_get_subject_name(x), X509_get_issuer_name(x)))
331:         x->ex_flags |= EXFLAG_SS;
332:     /* V1 should mean no extensions ... */
333:     if(!X509_get_version(x)) x->ex_flags |= EXFLAG_V1;
334:     /* Handle basic constraints */
335:     if((bs=X509_get_ext_d2i(x, NID_basic_constraints, NULL, NULL))) {
336:         if(bs->ca) x->ex_flags |= EXFLAG_CA;
337:         if(bs->pathlen) {
338:             if((bs->pathlen->type == V_ASN1_NEG_INTEGER)
339:                 || !bs->ca) {
340:                 x->ex_flags |= EXFLAG_INVALID;
341:                 x->ex_pathlen = 0;
342:             } else x->ex_pathlen = ASN1_INTEGER_get(bs->pathlen);
343:         } else x->ex_pathlen = -1;
344:         BASIC_CONSTRAINTS_free(bs);
345:         x->ex_flags |= EXFLAG_BCONS;
346:     }
347:     /* Handle RFC3820 proxy certificates */
348:     if((pci=X509_get_ext_d2i(x, NID_proxyCertInfo, NULL, NULL))) {
349:         if (x->ex_flags & EXFLAG_CA
350:             || X509_get_ext_by_NID(x, NID_subject_alt_name, 0) >= 0
351:             || X509_get_ext_by_NID(x, NID_issuer_alt_name, 0) >= 0) {
352:             x->ex_flags |= EXFLAG_INVALID;
353:         }
354:         if (pci->pcPathLengthConstraint) {
355:             x->ex_pcpPathlen =
356:                 ASN1_INTEGER_get(pci->pcPathLengthConstraint);
357:         } else x->ex_pcpPathlen = -1;
358:         PROXY_CERT_INFO_EXTENSION_free(pci);
359:         x->ex_flags |= EXFLAG_PROXY;
360:     }
361:     /* Handle Globus GSI-3 proxy certificates */
362:     if((pci_old=X509_get_ext_d2i(x, NID_globusProxyCertInfo, NULL, NULL))) {
363:         if (x->ex_flags & EXFLAG_CA
364:             || X509_get_ext_by_NID(x, NID_subject_alt_name, 0) >= 0
365:             || X509_get_ext_by_NID(x, NID_issuer_alt_name, 0) >= 0) {
366:             x->ex_flags |= EXFLAG_INVALID;
367:         }
368:         if (pci_old->path_length) {
369:             x->ex_pcpPathlen =
370:                 ASN1_INTEGER_get(pci_old->path_length);
371:         } else x->ex_pcpPathlen = -1;
372:         PROXYCERTINFO_free(pci_old);
373:         x->ex_flags |= EXFLAG_PROXY;
```

```
374: }
375: /* Handle key usage */
376: if((usage=X509_get_ext_d2i(x, NID_key_usage, NULL, NULL))) {
377:     if(usage->length > 0) {
378:         x->ex_kusage = usage->data[0];
379:         if(usage->length > 1)
380:             x->ex_kusage |= usage->data[1] << 8;
381:     } else x->ex_kusage = 0;
382:     x->ex_flags |= EXFLAG_KUSAGE;
383:     ASN1_BIT_STRING_free(usage);
384: }
385: x->ex_xkusage = 0;
386: if((extusage=X509_get_ext_d2i(x, NID_ext_key_usage, NULL, NULL))) {
387:     x->ex_flags |= EXFLAG_XKUSAGE;
388:     for(i = 0; i < sk ASN1_OBJECT_num(extusage); i++) {
389:         switch(OBJ_obj2nid(sk ASN1_OBJECT_value(extusage,i))) {
390:             case NID_server_auth:
391:                 x->ex_xkusage |= XCU_SSL_SERVER;
392:                 break;
393:
394:             case NID_client_auth:
395:                 x->ex_xkusage |= XCU_SSL_CLIENT;
396:                 break;
397:
398:             case NID_email_protect:
399:                 x->ex_xkusage |= XCU_SMIME;
400:                 break;
401:
402:             case NID_code_sign:
403:                 x->ex_xkusage |= XCU_CODE_SIGN;
404:                 break;
405:
406:             case NID_ms_sgc:
407:             case NID_ns_sgc:
408:                 x->ex_xkusage |= XCU_SGC;
409:                 break;
410:
411:             case NID_OCSP_sign:
412:                 x->ex_xkusage |= XCU_OCSP_SIGN;
413:                 break;
414:
415:             case NID_time_stamp:
416:                 x->ex_xkusage |= XCU_TIMESTAMP;
417:                 break;
418:
419:             case NID_dvcs:
420:                 x->ex_xkusage |= XCU_DVCS;
421:                 break;
422:         }
423:     }
424:     sk ASN1_OBJECT_pop_free(extusage, ASN1_OBJECT_free);
425: }
426:
427: if((ns=X509_get_ext_d2i(x, NID_netscape_cert_type, NULL, NULL))) {
428:     if(ns->length > 0) x->ex_nscert = ns->data[0];
429:     else x->ex_nscert = 0;
430:     x->ex_flags |= EXFLAG_NSCERT;
431:     ASN1_BIT_STRING_free(ns);
432: }
433: x->skid =X509_get_ext_d2i(x, NID_subject_key_identifier, NULL, NULL);
434: x->akid =X509_get_ext_d2i(x, NID_authority_key_identifier, NULL, NULL);
435: #ifndef OPENSSL_NO_RFC3779
436: x->rfc3779_addr =X509_get_ext_d2i(x, NID_sbgp_ipAddrBlock, NULL, NULL);
437: x->rfc3779_asid =X509_get_ext_d2i(x, NID_sbgp_autonomousSysNum,
438:                                         NULL, NULL);
```

```
439: #endif
440:     for (i = 0; i < X509_get_ext_count(x); i++)
441:     {
442:         ex = X509_get_ext(x, i);
443:         if (!X509_EXTENSION_get_critical(ex))
444:             continue;
445:         if (!X509_supported_extension(ex))
446:             {
447:                 x->ex_flags |= EXFLAG_CRITICAL;
448:                 break;
449:             }
450:     }
451:     x->ex_flags |= EXFLAG_SET;
452: }
453:
454: /* CA checks common to all purposes
455:  * return codes:
456:  * 0 not a CA
457:  * 1 is a CA
458:  * 2 basicConstraints absent so "maybe" a CA
459:  * 3 basicConstraints absent but self signed V1.
460:  * 4 basicConstraints absent but keyUsage present and keyCertSign asserted.
461: */
462:
463: #define V1_ROOT (EXFLAG_V1|EXFLAG_SS)
464: #define ku_reject(x, usage) \
465:     (((x)->ex_flags & EXFLAG_KUSAGE) && !((x)->ex_kusage & (usage)))
466: #define xku_reject(x, usage) \
467:     (((x)->ex_flags & EXFLAG_XKUSAGE) && !((x)->ex_xkusage & (usage)))
468: #define ns_reject(x, usage) \
469:     (((x)->ex_flags & EXFLAG_NSCERT) && !((x)->ex_nscert & (usage)))
470:
471: static int check_ca(const X509 *x)
472: {
473:     /* keyUsage if present should allow cert signing */
474:     if(ku_reject(x, KU_KEY_CERT_SIGN)) return 0;
475:     if(x->ex_flags & EXFLAG_BCONS) {
476:         if(x->ex_flags & EXFLAG_CA) return 1;
477:         /* If basicConstraints says not a CA then say so */
478:         else return 0;
479:     } else {
480:         /* we support V1 roots for... uh, I don't really know why. */
481:         if((x->ex_flags & V1_ROOT) == V1_ROOT) return 3;
482:         /* If key usage present it must have certSign so tolerate it */
483:         else if (x->ex_flags & EXFLAG_KUSAGE) return 4;
484:         /* Older certificates could have Netscape-specific CA types */
485:         else if (x->ex_flags & EXFLAG_NSCERT
486:                 && x->ex_nscert & NS_ANY_CA) return 5;
487:         /* can this still be regarded a CA certificate? I doubt it */
488:         return 0;
489:     }
490: }
491:
492: int X509_check_ca(X509 *x)
493: {
494:     if(!(x->ex_flags & EXFLAG_SET)) {
495:         CRYPTO_w_lock(CRYPTO_LOCK_X509);
496:         x509v3_cache_extensions(x);
497:         CRYPTO_w_unlock(CRYPTO_LOCK_X509);
498:     }
499:
500:     return check_ca(x);
501: }
502:
503: /* Check SSL CA: common checks for SSL client and server */
```

```

v3_purp.c      Mon Apr 30 19:22:42 2007      9

504: static int check_ssl_ca(const X509 *x)
505: {
506:     int ca_ret;
507:     ca_ret = check_ca(x);
508:     if(!ca_ret) return 0;
509:     /* check nsCertType if present */
510:     if(ca_ret != 5 || x->ex_nscert & NS_SSL_CA) return ca_ret;
511:     else return 0;
512: }
513:
514:
515: static int check_purpose_ssl_client(const X509_PURPOSE *xp, const X509 *x, int c
a)
516: {
517:     if(xku_reject(x,XKU_SSL_CLIENT)) return 0;
518:     if(ca) return check_ssl_ca(x);
519:     /* We need to do digital signatures with it */
520:     if(ku_reject(x,KU_DIGITAL_SIGNATURE)) return 0;
521:     /* nsCertType if present should allow SSL client use */
522:     if(ns_reject(x, NS_SSL_CLIENT)) return 0;
523:     return 1;
524: }
525:
526: static int check_purpose_ssl_server(const X509_PURPOSE *xp, const X509 *x, int c
a)
527: {
528:     if(xku_reject(x,XKU_SSL_SERVER|XKU_SGC)) return 0;
529:     if(ca) return check_ssl_ca(x);
530:
531:     if(ns_reject(x, NS_SSL_SERVER)) return 0;
532:     /* Now as for keyUsage: we'll at least need to sign OR encipher */
533:     if(ku_reject(x, KU_DIGITAL_SIGNATURE|KU_KEY_ENCIPHERMENT)) return 0;
534:
535:     return 1;
536:
537: }
538:
539: static int check_purpose_ns_ssl_server(const X509_PURPOSE *xp, const X509 *x, in
t ca)
540: {
541:     int ret;
542:     ret = check_purpose_ssl_server(xp, x, ca);
543:     if(!ret || ca) return ret;
544:     /* We need to encipher or Netscape complains */
545:     if(ku_reject(x, KU_KEY_ENCIPHERMENT)) return 0;
546:     return ret;
547: }
548:
549: /* common S/MIME checks */
550: static int purpose_smime(const X509 *x, int ca)
551: {
552:     if(xku_reject(x,XKU_SMIME)) return 0;
553:     if(ca) {
554:         int ca_ret;
555:         ca_ret = check_ca(x);
556:         if(!ca_ret) return 0;
557:         /* check nsCertType if present */
558:         if(ca_ret != 5 || x->ex_nscert & NS_SMIME_CA) return ca_ret;
559:         else return 0;
560:     }
561:     if(x->ex_flags & EXFLAG_NSCERT) {
562:         if(x->ex_nscert & NS_SMIME) return 1;
563:         /* Workaround for some buggy certificates */
564:         if(x->ex_nscert & NS_SSL_CLIENT) return 2;
565:     }

```

```

v3_purp.c      Mon Apr 30 19:22:42 2007      10

566:     }
567:     return 1;
568: }
569:
570: static int check_purpose_smime_sign(const X509_PURPOSE *xp, const X509 *x, int c
a)
571: {
572:     int ret;
573:     ret = purpose_smime(x, ca);
574:     if(!ret || ca) return ret;
575:     if(ku_reject(x, KU_DIGITAL_SIGNATURE|KU_NON_REPUDIATION)) return 0;
576:     return ret;
577: }
578:
579: static int check_purpose_smime_encrypt(const X509_PURPOSE *xp, const X509 *x, in
t ca)
580: {
581:     int ret;
582:     ret = purpose_smime(x, ca);
583:     if(!ret || ca) return ret;
584:     if(ku_reject(x, KU_KEY_ENCIPHERMENT)) return 0;
585:     return ret;
586: }
587:
588: static int check_purpose_crl_sign(const X509_PURPOSE *xp, const X509 *x, int ca)
589: {
590:     if(ca) {
591:         int ca_ret;
592:         if((ca_ret = check_ca(x)) != 2) return ca_ret;
593:         else return 0;
594:     }
595:     if(ku_reject(x, KU_CRL_SIGN)) return 0;
596:     return 1;
597: }
598:
599: /* OCSP helper: this is *not* a full OCSP check. It just checks that
600: * each CA is valid. Additional checks must be made on the chain.
601: */
602:
603: static int ocsp_helper(const X509_PURPOSE *xp, const X509 *x, int ca)
604: {
605:     /* Must be a valid CA. Should we really support the "I don't know"
606:      value (2)? */
607:     if(ca) return check_ca(x);
608:     /* leaf certificate is checked in OCSP_verify() */
609:     return 1;
610: }
611:
612: static int no_check(const X509_PURPOSE *xp, const X509 *x, int ca)
613: {
614:     return 1;
615: }
616:
617: /* Various checks to see if one certificate issued the second.
618: * This can be used to prune a set of possible issuer certificates
619: * which have been looked up using some simple method such as by
620: * subject name.
621: * These are:
622: * 1. Check issuer_name(subject) == subject_name(issuer)
623: * 2. If akid(subject) exists check it matches issuer
624: * 3. If key_usage(issuer) exists check it supports certificate signing
625: * returns 0 for OK, positive for reason for mismatch, reasons match
626: * codes for X509_verify_cert()
627: */
628:

```

```

v3_purp.c      Mon Apr 30 19:22:42 2007      11
629: int X509_check_issued(X509 *issuer, X509 *subject)
630: {
631:     if(X509_NAME_cmp(X509_get_subject_name(issuer),
632:                         X509_get_issuer_name(subject)))
633:         return X509_V_ERR_SUBJECT_ISSUER_MISMATCH;
634:     x509v3_cache_extensions(issuer);
635:     x509v3_cache_extensions(subject);
636:     if(subject->akid) {
637:         /* Check key ids (if present) */
638:         if(subject->akid->keyid && issuer->skid &&
639:             ASN1_OCTET_STRING_cmp(subject->akid->keyid, issuer->skid) )
640:             return X509_V_ERR_AKID_SKID_MISMATCH;
641:         /* Check serial number */
642:         if(subject->akid->serial &&
643:             ASN1_INTEGER_cmp(X509_get_serialNumber(issuer),
644:                             subject->akid->serial))
645:             return X509_V_ERR_AKID_ISSUER_SERIAL_MISMATCH;
646:         /* Check issuer name */
647:         if(subject->akid->issuer) {
648:             /* Ugh, for some peculiar reason AKID includes
649:              * SEQUENCE OF GeneralName. So look for a DirName.
650:              * There may be more than one but we only take any
651:              * notice of the first.
652:             */
653:             GENERAL_NAMES *gens;
654:             GENERAL_NAME *gen;
655:             X509_NAME *nm = NULL;
656:             int i;
657:             gens = subject->akid->issuer;
658:             for(i = 0; i < sk_GENERAL_NAME_num(gens); i++) {
659:                 gen = sk_GENERAL_NAME_value(gens, i);
660:                 if(gen->type == GEN_DIRNAME) {
661:                     nm = gen->d.dirn;
662:                     break;
663:                 }
664:             }
665:             if(nm && X509_NAME_cmp(nm, X509_get_issuer_name(issuer)))
666:                 return X509_V_ERR_AKID_ISSUER_SERIAL_MISMATCH;
667:         }
668:     }
669:     /* Check for "old" proxy cert: Globus legacy or Globus pre-RFC
670:        proxy cert. */
671:     if((subject->ex_flags & EXFLAG_PROXY) || is_old_globus_proxy(subject))
672:     {
673:         if(ku_reject(issuer, KU_DIGITAL_SIGNATURE))
674:             return X509_V_ERR_KEYUSAGE_NO_DIGITAL_SIGNATURE;
675:     }
676:     else if(ku_reject(issuer, KU_KEY_CERT_SIGN))
677:         return X509_V_ERR_KEYUSAGE_NO_CERTSIGN;
678:     return X509_V_OK;
679: }
680:

```

Attachment 6

globus_proxycert.c

```
1: /* crypto/x509/globus_proxycert.c */
2:
3: #include <stdio.h>
4: #include "globus_proxycert.h"
5:
6: /**
7:  * Certificate Types, copied from Globus Toolkit 4 (globus-4.0.3)
8:  * file globus_gsi_cert_utils_constants.h
9: */
10: typedef enum globus_gsi_cert_utils_cert_type_e
11: {
12:     /** A end entity certificate */
13:     GLOBUS_GSI_CERT_UTILS_TYPE_EEC,
14:     /** A CA certificate */
15:     GLOBUS_GSI_CERT_UTILS_TYPE_CA,
16:     /** A X.509 Proxy Certificate Profile (pre-RFC) compliant
17:      * impersonation proxy
18:      */
19:     GLOBUS_GSI_CERT_UTILS_TYPE_GSI_3_IMPERSONATION_PROXY,
20:     /** A X.509 Proxy Certificate Profile (pre-RFC) compliant
21:      * independent proxy
22:      */
23:     GLOBUS_GSI_CERT_UTILS_TYPE_GSI_3_INDEPENDENT_PROXY,
24:     /** A X.509 Proxy Certificate Profile (pre-RFC) compliant
25:      * limited proxy
26:      */
27:     GLOBUS_GSI_CERT_UTILS_TYPE_GSI_3_LIMITED_PROXY,
28:     /** A X.509 Proxy Certificate Profile (pre-RFC) compliant
29:      * restricted proxy
30:      */
31:     GLOBUS_GSI_CERT_UTILS_TYPE_GSI_3_RESTRICTED_PROXY,
32:     /** A legacy Globus impersonation proxy */
33:     GLOBUS_GSI_CERT_UTILS_TYPE_GSI_2_PROXY,
34:     /** A legacy Globus limited impersonation proxy */
35:     GLOBUS_GSI_CERT_UTILS_TYPE_GSI_2_LIMITED_PROXY,
36:     /** A X.509 Proxy Certificate Profile RFC compliant impersonation proxy */
37:     GLOBUS_GSI_CERT_UTILS_TYPE_RFC_IMPERSONATION_PROXY,
38:     /** A X.509 Proxy Certificate Profile RFC compliant independent proxy */
39:     GLOBUS_GSI_CERT_UTILS_TYPE_RFC_INDEPENDENT_PROXY,
40:     /** A X.509 Proxy Certificate Profile RFC compliant limited proxy */
41:     GLOBUS_GSI_CERT_UTILS_TYPE_RFC_LIMITED_PROXY,
42:     /** A X.509 Proxy Certificate Profile RFC compliant restricted proxy */
43:     GLOBUS_GSI_CERT_UTILS_TYPE_RFC_RESTRICTED_PROXY
44: } globus_gsi_cert_utils_cert_type_t;
45:
46: /**
47:  * Certificate Type macros, copied from Globus Toolkit 4 (globus-4.0.3)
48:  * file globus_gsi_cert_utils.h
49: */
50: #define GLOBUS_GSI_CERT_UTILS_IS_PROXY(cert_type) \
51:     (cert_type == GLOBUS_GSI_CERT_UTILS_TYPE_GSI_3_IMPERSONATION_PROXY || \
52:      cert_type == GLOBUS_GSI_CERT_UTILS_TYPE_GSI_3_INDEPENDENT_PROXY || \
53:      cert_type == GLOBUS_GSI_CERT_UTILS_TYPE_GSI_3_LIMITED_PROXY || \
54:      cert_type == GLOBUS_GSI_CERT_UTILS_TYPE_GSI_3_RESTRICTED_PROXY || \
55:      cert_type == GLOBUS_GSI_CERT_UTILS_TYPE_RFC_IMPERSONATION_PROXY || \
56:      cert_type == GLOBUS_GSI_CERT_UTILS_TYPE_RFC_INDEPENDENT_PROXY || \
57:      cert_type == GLOBUS_GSI_CERT_UTILS_TYPE_RFC_LIMITED_PROXY || \
58:      cert_type == GLOBUS_GSI_CERT_UTILS_TYPE_RFC_RESTRICTED_PROXY || \
59:      cert_type == GLOBUS_GSI_CERT_UTILS_TYPE_GSI_2_PROXY || \
60:      cert_type == GLOBUS_GSI_CERT_UTILS_TYPE_GSI_2_LIMITED_PROXY)
61:
62: #define GLOBUS_GSI_CERT_UTILS_IS_GSI_3_PROXY(cert_type) \
63:     (cert_type == GLOBUS_GSI_CERT_UTILS_TYPE_GSI_3_IMPERSONATION_PROXY || \
64:      cert_type == GLOBUS_GSI_CERT_UTILS_TYPE_GSI_3_INDEPENDENT_PROXY || \
65:      cert_type == GLOBUS_GSI_CERT_UTILS_TYPE_GSI_3_LIMITED_PROXY || \
```

```
66:             cert_type == GLOBUS_GSI_CERT_UTILS_TYPE_GSI_3_RESTRICTED_PROXY)
67:
68: #define GLOBUS_GSI_CERT_UTILS_IS_GSI_2_PROXY(cert_type) \
69:     (cert_type == GLOBUS_GSI_CERT_UTILS_TYPE_GSI_2_PROXY || \
70:      cert_type == GLOBUS_GSI_CERT_UTILS_TYPE_GSI_2_LIMITED_PROXY)
71:
72: #define GLOBUS_SUCCESS 0
73:
74: /* Compute the Globus certificate type for the supplied certificate.
75: * Copied almost verbatim from globus_gsi_cert_utils_get_cert_type()
76: * in Globus Toolkit 4 (globus-4.0.3), file:
77: *     source-trees/gsi/cert_utils/source/library/globus_gsi_cert_utils.c
78: * The only significant difference is that Globus error reporting
79: * has been commented out - it should be replaced with OpenSSL error
80: * reporting.
81: *
82: * Returns GLOBUS_SUCCESS on success (meaning that a valid value is
83: * passed back in the 'type' parameter), some error value on failure.
84: */
85: static int get_globus_cert_type(
86:     X509                                * cert,
87:     globus_gsi_cert_utils_cert_type_t * type)
88: {
89:     X509_NAME                * subject = NULL;
90:     X509_NAME                * name = NULL;
91:     X509_NAME_ENTRY          * ne = NULL;
92:     X509_NAME_ENTRY          * new_ne = NULL;
93:     X509_EXTENSION           * pci_ext = NULL;
94:     ASN1_STRING               * data = NULL;
95:     PROXYCERTINFO            * pci = NULL;
96:     PROXYPOLICY              * policy = NULL;
97:     ASN1_OBJECT               * policy_lang = NULL;
98:     int                      policy_nid;
99:     int                      result = GLOBUS_SUCCESS;
100:    int                      index = -1;
101:    int                      critical;
102:    BASIC_CONSTRAINTS        * x509v3_bc = NULL;
103:
104:    /* assume cert is an EEC if nothing else matches */
105:    *type = GLOBUS_GSI_CERT_UTILS_TYPE_EEC;
106:
107:    if((x509v3_bc = X509_get_ext_d2i(cert,
108:                                         NID_basic_constraints,
109:                                         &critical,
110:                                         &index)) && x509v3_bc->ca)
111:    {
112:        *type = GLOBUS_GSI_CERT_UTILS_TYPE_CA;
113:        goto exit;
114:    }
115:
116:    subject = X509_get_subject_name(cert);
117:
118:    if((ne = X509_NAME_get_entry(subject, X509_NAME_entry_count(subject)-1))
119:       == NULL)
120:    {
121:        result = -1;
122:        /*
123:         GLOBUS_GSI_CERT_UTILS_OPENSSL_ERROR_RESULT(
124:             result,
125:             GLOBUS_GSI_CERT_UTILS_ERROR_GETTING_NAME_ENTRY_OF SUBJECT,
126:             (_CUSL("Can't get X509 name entry from subject")));
127:         */
128:        goto exit;
129:    }
130:
```

```
131:     if (!OBJ_cmp(ne->object, OBJ_nid2obj(NID_commonName)))
132:     {
133:         /* the name entry is of the type: common name */
134:         data = X509_NAME_ENTRY_get_data(ne);
135:         if(data->length == 5 && !memcmp(data->data, "proxy", 5))
136:         {
137:             *type = GLOBUS_GSI_CERT_UTILS_TYPE_GSI_2_PROXY;
138:         }
139:         else if(data->length == 13 && !memcmp(data->data, "limited proxy", 13))
140:         {
141:             *type = GLOBUS_GSI_CERT_UTILS_TYPE_GSI_2_LIMITED_PROXY;
142:         }
143:         else if((index = X509_get_ext_by_NID(cert,
144:                                         OBJ_sn2nid(PROXYCERTINFO_SN),
145:                                         -1)) != -1 &&
146:                 (pci_ext = X509_get_ext(cert, index)) &&
147:                 X509_EXTENSION_get_critical(pci_ext))
148:         {
149:             if((pci = X509V3_EXT_d2i(pci_ext)) == NULL)
150:             {
151:                 result = -2;
152:                 /*
153:                  GLOBUS_GSI_CERT_UTILS_OPENSSL_ERROR_RESULT(
154:                      result,
155:                      GLOBUS_GSI_CERT_UTILS_ERROR_NON_COMPLIANT_PROXY,
156:                      (_CUSL("Can't convert DER encoded PROXYCERTINFO "
157:                            "extension to internal form")));
158:                 */
159:                 goto exit;
160:             }
161:
162:             if((policy = PROXYCERTINFO_get_policy(pci)) == NULL)
163:             {
164:                 result = -3;
165:                 /*
166:                  GLOBUS_GSI_CERT_UTILS_OPENSSL_ERROR_RESULT(
167:                      result,
168:                      GLOBUS_GSI_CERT_UTILS_ERROR_NON_COMPLIANT_PROXY,
169:                      (_CUSL("Can't get policy from PROXYCERTINFO extension")));
170:                 */
171:                 goto exit;
172:             }
173:
174:             if((policy_lang = PROXPOLICY_get_policy_language(policy))
175:                == NULL)
176:             {
177:                 result = -4;
178:                 /*
179:                  GLOBUS_GSI_CERT_UTILS_OPENSSL_ERROR_RESULT(
180:                      result,
181:                      GLOBUS_GSI_CERT_UTILS_ERROR_NON_COMPLIANT_PROXY,
182:                      (_CUSL("Can't get policy language from"
183:                            " PROXYCERTINFO extension")));
184:                 */
185:                 goto exit;
186:             }
187:
188:             policy_nid = OBJ_obj2nid(policy_lang);
189:
190:             if(policy_nid == OBJ_sn2nid(IMPERSONATION_PROXY_SN))
191:             {
192:                 *type = GLOBUS_GSI_CERT_UTILS_TYPE_RFC_IMPERSONATION_PROXY;
193:             }
194:             else if(policy_nid == OBJ_sn2nid(INDEPENDENT_PROXY_SN))
195:             {
```

```
196:             *type = GLOBUS_GSI_CERT_UTILS_TYPE_RFC_INDEPENDENT_PROXY;
197:         }
198:     else if(policy_nid == OBJ_sn2nid(LIMITED_PROXY_SN))
199:     {
200:         *type = GLOBUS_GSI_CERT_UTILS_TYPE_RFC_LIMITED_PROXY;
201:     }
202:     else
203:     {
204:         *type = GLOBUS_GSI_CERT_UTILS_TYPE_RFC_RESTRICTED_PROXY;
205:     }
206:
207:     if(X509_get_ext_by_NID(cert,
208:                             OBJ_sn2nid(PROXYCERTINFO_SN),
209:                             index) != -1)
210:     {
211:         result = -5;
212:         /*
213:          GLOBUS_GSI_CERT_UTILS_OPENSSL_ERROR_RESULT(
214:              result,
215:              GLOBUS_GSI_CERT_UTILS_ERROR_NON_COMPLIANT_PROXY,
216:              (_CUSL("Found more than one PCI extension")));
217:         */
218:         goto exit;
219:     }
220: }
221: else if((index = X509_get_ext_by_NID(cert,
222:                                         OBJ_sn2nid(PROXYCERTINFO_OLD_SN),
223:                                         -1)) != -1 &&
224:          (pci_ext = X509_get_ext(cert,index)) &&
225:          X509_EXTENSION_get_critical(pci_ext))
226: {
227:     if((pci = X509V3_EXT_d2i(pci_ext)) == NULL)
228:     {
229:         result = -6;
230:         /*
231:          GLOBUS_GSI_CERT_UTILS_OPENSSL_ERROR_RESULT(
232:              result,
233:              GLOBUS_GSI_CERT_UTILS_ERROR_NON_COMPLIANT_PROXY,
234:              (_CUSL("Can't convert DER encoded PROXYCERTINFO "
235:                  "extension to internal form")));
236:         */
237:         goto exit;
238:     }
239:
240:     if((policy = PROXYCERTINFO_get_policy(pci)) == NULL)
241:     {
242:         result = -7;
243:         /*
244:          GLOBUS_GSI_CERT_UTILS_OPENSSL_ERROR_RESULT(
245:              result,
246:              GLOBUS_GSI_CERT_UTILS_ERROR_NON_COMPLIANT_PROXY,
247:              (_CUSL("Can't get policy from PROXYCERTINFO extension")));
248:         */
249:         goto exit;
250:     }
251:
252:     if((policy_lang = PROXYPOLICY_get_policy_language(policy))
253:        == NULL)
254:     {
255:         result = -8;
256:         /*
257:          GLOBUS_GSI_CERT_UTILS_OPENSSL_ERROR_RESULT(
258:              result,
259:              GLOBUS_GSI_CERT_UTILS_ERROR_NON_COMPLIANT_PROXY,
260:              (_CUSL("Can't get policy language from"
```

```

globus_proxycert.c      Wed Apr 25 10:57:41 2007      5
261:                     " PROXYCERTINFO extension" ))));
262:     */
263:     goto exit;
264: }
265:
266: policy_nid = OBJ_obj2nid(policy_lang);
267:
268: if(policy_nid == OBJ_sn2nid(IMPERSONATION_PROXY_SN))
269: {
270:     *type = GLOBUS_GSI_CERT_UTILS_TYPE_GSI_3_IMPERSONATION_PROXY;
271: }
272: else if(policy_nid == OBJ_sn2nid(INDEPENDENT_PROXY_SN))
273: {
274:     *type = GLOBUS_GSI_CERT_UTILS_TYPE_GSI_3_INDEPENDENT_PROXY;
275: }
276: else if(policy_nid == OBJ_sn2nid(LIMITED_PROXY_SN))
277: {
278:     *type = GLOBUS_GSI_CERT_UTILS_TYPE_GSI_3_LIMITED_PROXY;
279: }
280: else
281: {
282:     *type = GLOBUS_GSI_CERT_UTILS_TYPE_GSI_3_RESTRICTED_PROXY;
283: }
284:
285: if(X509_get_ext_by_NID(cert,
286:                         OBJ_sn2nid(PROXYCERTINFO_OLD_SN),
287:                         index) != -1)
288: {
289:     result = -9;
290:     /*
291:      GLOBUS_GSI_CERT_UTILS_OPENSSL_ERROR_RESULT(
292:          result,
293:          GLOBUS_GSI_CERT_UTILS_ERROR_NON_COMPLIANT_PROXY,
294:          (_CUSL("Found more than one PCI extension")));
295:     */
296:     goto exit;
297: }
298:
299:
300: if(GLOBUS_GSI_CERT_UTILS_IS_PROXY(*type))
301: {
302:     /* it is some kind of proxy - now we check if the subject
303:      * matches the signer, by adding the proxy name entry CN
304:      * to the signer's subject
305:     */
306:     if((name = X509_NAME_dup(
307:                 X509_get_issuer_name(cert))) == NULL)
308:     {
309:         result = -10;
310:         /*
311:          GLOBUS_GSI_CERT_UTILS_OPENSSL_ERROR_RESULT(
312:              result,
313:              GLOBUS_GSI_CERT_UTILS_ERROR COPYING SUBJECT,
314:              (_CUSL("Error copying X509_NAME struct")));
315:         */
316:         goto exit;
317:     }
318:
319:     if((new_ne = X509_NAME_ENTRY_create_by_NID(NULL, NID_commonName,
320:                                                 V_ASN1_APP_CHOOSE,
321:                                                 data->data, -1)) == NULL)
322:     {
323:         result = -11;
324:         /*
325:          GLOBUS_GSI_CERT_UTILS_OPENSSL_ERROR_RESULT(

```

```
326:                     result,
327:                     GLOBUS_GSI_CERT_UTILS_ERROR_GETTING_CN_ENTRY,
328:                     (_CUSL("Error creating X509 name entry of: %s"), data->data)
);
329:     */
330:     goto exit;
331: }
332:
333: if(!X509_NAME_add_entry(name, new_ne, X509_NAME_entry_count(name),0)
)
334: {
335:     X509_NAME_ENTRY_free(new_ne);
336:     new_ne = NULL;
337:     result = -12;
338:     /*
339:      GLOBUS_GSI_CERT_UTILS_OPENSSL_ERROR_RESULT(
340:          result,
341:          GLOBUS_GSI_CERT_UTILS_ERROR_ADDING_CN_TO SUBJECT,
342:          (_CUSL("Error adding name entry with value: %s, to subject"))

343:          data->data));
344:     */
345:     goto exit;
346: }
347:
348: if(new_ne)
{
349:     X509_NAME_ENTRY_free(new_ne);
350:     new_ne = NULL;
351: }
352:
353:
354: if (X509_NAME_cmp(name,subject))
{
355:
356:     /*
357:      * Reject this certificate, only the user
358:      * may sign the proxy
359:      */
360:     result = -13;
361:     /*
362:      GLOBUS_GSI_CERT_UTILS_OPENSSL_ERROR_RESULT(
363:          result,
364:          GLOBUS_GSI_CERT_UTILS_ERROR_NON_COMPLIANT_PROXY,
365:          (_CUSL("Issuer name + proxy CN entry does not equal subject
name")));
366:     */
367:     goto exit;
368: }
369:
370: if(name)
{
371:     X509_NAME_free(name);
372:     name = NULL;
373: }
374:
375: }
376: }
377:
378: result = GLOBUS_SUCCESS;
379:
380: exit:
381:
382: if(x509v3_bc)
{
383:     BASIC_CONSTRAINTS_free(x509v3_bc);
384:
385: }
```

```
387:     if(new_ne)
388:     {
389:         X509_NAME_ENTRY_free(new_ne);
390:     }
391:
392:     if(name)
393:     {
394:         X509_NAME_free(name);
395:     }
396:
397:     if(pci)
398:     {
399:         PROXYCERTINFO_free(pci);
400:     }
401:
402:     return result;
403: }
404:
405: /*
406: * Check that a "Globus (GSI 3) proxyCertInfo extension" Object has
407: * been created, and create it if it doesn't already exist. This
408: * object is needed for lookups in this file and in ../x509v3/v3_purp.c.
409: * The object could have been initialized by modifying files in
410: * ../objects; it is initialized here to minimize changes to OpenSSL code.
411: * It is essential that the object be initialized before it is
412: * needed in v3_purp.c.
413: * The init code (run in first call to function) is copied from GT4 file:
414: *      source-trees/gsi/openssl_module/source/library/globus_openssl.c
415: *
416: * Returns:
417: *      1 if object has been successfully created
418: *      0 on any error
419: */
420: int check_globus_init()
421: {
422:     static int firstcall = 1;
423:     if(firstcall) {
424:         firstcall = 0;
425:         int old_pci_NID = OBJ_create(PROXYCERTINFO_OLD_OID,
426:                                         PROXYCERTINFO_OLD_SN,
427:                                         PROXYCERTINFO_OLD_LN);
428:         if(old_pci_NID != NID_globusProxyCertInfo) {
429:             return 0;
430:         }
431:         X509V3_EXT_METHOD * pci_old_x509v3_ext_meth =
432:             PROXYCERTINFO_OLD_x509v3_ext_meth();
433:         pci_old_x509v3_ext_meth->ext_nid = old_pci_NID;
434:         X509V3_EXT_add(pci_old_x509v3_ext_meth);
435:     }
436:     return 1;
437: }
438:
439: /*
440: * Check for two flavors of pre-RFC3820 proxy certs:
441: * 1. Globus "old" proxy cert (GSI 2).
442: * 2. Globus "pre-RFC" proxy cert (GSI 3).
443: *
444: * Returns:
445: *      1 if cert is an old/pre-RFC proxy certificate.
446: *      0 if cert is not an old/pre-RFC proxy certificate.
447: */
448: int is_old_globus_proxy(X509 * cert)
449: {
450:     int ret = 0;
451:
```

```
452: /* Compute certificate type for cert. */
453: globus_gsi_cert_utils_cert_type_t cert_type;
454: cert_type = GLOBUS_GSI_CERT_UTILS_TYPE_EEC;
455: int result = get_globus_cert_type(cert, &cert_type);
456: if (result != GLOBUS_SUCCESS)
457: {
458:     fprintf(stderr, "get_globus_cert_type error result: %d\n", result);
459: }
460: else
461: {
462:     if (GLOBUS_GSI_CERT_UTILS_IS_GSI_2_PROXY(cert_type) ||
463:         GLOBUS_GSI_CERT_UTILS_IS_GSI_3_PROXY(cert_type))
464:     {
465:         ret = 1;
466:     }
467: }
468: return(ret);
469: }
470:
```

Attachment 7

globus_proxycert.h

```
1: /* crypto/x509/globus_proxycert.h */
2:
3: #ifndef HEADER_GLOBUS_PROXYCERT_H
4: #define HEADER_GLOBUS_PROXYCERT_H
5:
6: #include "proxycertinfo.h"
7:
8: #ifdef __cplusplus
9: extern "C" {
10: #endif
11:
12: int check_globus_init();
13: int is_old_globus_proxy(X509 *cert);
14:
15: /* Set NID for globuxProxyCertInfo object to one more than
16:  * largest NID value in ../objects/obj_mac.h, which is 771.
17:  * Extracted with this command:
18:  *      grep NID ../objects/obj_mac.h | sort -n -k 3| tail -1 |cut -f 3
19:  */
20: #define NID_globusProxyCertInfo 772
21:
22: #ifdef __cplusplus
23: }
24: #endif
25: #endif
```

Attachment 8

proxycertinfo.c

```
1: /*
2:  * Portions of this file Copyright 1999-2005 University of Chicago
3:  * Portions of this file Copyright 1999-2005 The University of Southern California
4:  */
5: /*
6:  * This file or a portion of this file is licensed under the
7:  * terms of the Globus Toolkit Public License, found at
8:  * http://www.globus.org/toolkit/download/license.html.
9:  * If you redistribute this file, with or without
10: * modifications, you must include this notice in the file.
11: */
12: #ifndef GLOBUS_DONT_DOCUMENT_INTERNAL
13: /**
14:  * @file proxycertinfo.c
15:  *
16:  * RCSfile: proxycertinfo.c,v $ 
17:  * Revision: 1.12 $ 
18:  * Date: 2005/04/16 00:09:55 $ 
19:  * Author: meder $ 
20:  */
21: #endif
22:
23: #include <stdio.h>
24: #include <openssl/err.h>
25: #include <openssl/asn1.h>
26: #include <openssl/asn1_mac.h>
27: #include <openssl/x509v3.h>
28:
29: #include "proxycertinfo.h"
30:
31: /**
32:  * @name ASN1_METHOD
33:  */
34: /*@{ */
35: /**
36:  * Define the functions required for
37:  * manipulating a PROXYCERTINFO and its ASN1 form.
38:  * @ingroup proxycertinfo
39:  *
40:  * Creates an ASN1_METHOD structure, which contains
41:  * pointers to routines that convert any PROXYCERTINFO
42:  * structure to its associated ASN1 DER encoded form
43:  * and vice-versa.
44:  *
45:  * @return the ASN1_METHOD object
46:  */
47: ASN1_METHOD * PROXYCERTINFO_asn1_meth()
48: {
49:     static ASN1_METHOD proxycertinfo_asn1_meth =
50:     {
51:         (int (*)()) i2d_PROXYCERTINFO,
52:         (char *(*)()) d2i_PROXYCERTINFO,
53:         (char *(*)()) PROXYCERTINFO_new,
54:         (void *(*)()) PROXYCERTINFO_free
55:     };
56:     return (&proxycertinfo_asn1_meth);
57: }
58: /* PROXYCERTINFO_asn1_meth() */
59: /*@}*/
60:
61: /**
62:  * @name New
63:  */
64: */
```

```
65: /*@{ */
66: /**
67:  * Create a new PROXYCERTINFO.
68:  * @ingroup proxycertinfo
69:  *
70:  * Allocates and initializes a new PROXYCERTINFO structure.
71:  *
72:  * @return pointer to the new PROXYCERTINFO
73:  */
74: PROXYCERTINFO * PROXYCERTINFO_new( )
75: {
76:     PROXYCERTINFO *                                ret;
77:     ASN1_CTX                                         c;
78:
79:     ret = NULL;
80:
81:     M_ASN1_New_Malloc(ret, PROXYCERTINFO);
82:     memset(ret, (int) NULL, sizeof(PROXYCERTINFO));
83:     ret->path_length      = NULL;
84:     ret->policy          = PROXYPOLICY_new();
85:     return (ret);
86:     M_ASN1_New_Error(ASN1_F_PROXYCERTINFO_NEW);
87: }
88: /* PROXYCERTINFO_new() */
89: /* @} */
90:
91:
92: /**
93:  * @name Free.
94:  */
95: /* @{ */
96: /**
97:  * Free a PROXYCERTINFO.
98:  * @ingroup proxycertinfo
99:  *
100: * @param cert_info pointer to the PROXYCERTINFO structure
101: * to be freed.
102: */
103: void PROXYCERTINFO_free(
104:     PROXYCERTINFO *                                cert_info)
105: {
106:     if(cert_info == NULL) return;
107:     ASN1_INTEGER_free(cert_info->path_length);
108:     PROXYPOLICY_free(cert_info->policy);
109:     OPENSSL_free(cert_info);
110: }
111: /* PROXYCERTINFO_free */
112: /* @} */
113:
114:
115: /**
116:  * @name Duplicate
117:  */
118: /* @{ */
119: /**
120:  * Makes a copy of a PROXYCERTINFO.
121:  * @ingroup proxycertinfo
122:  *
123:  * Makes a copy of a PROXYCERTINFO structure
124:  *
125:  * @param cert_info the PROXYCERTINFO structure to copy
126:  *
127:  * @return the copied PROXYCERTINFO structure
128:  */
129: PROXYCERTINFO * PROXYCERTINFO_dup(
```

```
130:     PROXYCERTINFO *                      cert_info)
131: {
132:     return ((PROXYCERTINFO *) ASN1_dup((int (*)())i2d_PROXYCERTINFO,
133:                                         (char *(*)())d2i_PROXYCERTINFO,
134:                                         (char *)cert_info));
135: }
136: /* PROXYCERTINFO_dup() */
137: /* @} */
138:
139: /**
140:  * @name Compare
141:  */
142: /* @{ */
143: /**
144:  * @ingroup proxycertinfo
145:  *
146:  * Compares two PROXYCERTINFO structures
147:  *
148:  * @param a pointer to the first PROXYCERTINFO structure
149:  * @param b pointer to the second PROXYCERTINFO structure
150:  *
151:  * @return an integer - the result of the comparison.
152:  * The comparison compares each of the fields, so if any of those
153:  * fields are not equal then a nonzero value is returned. Equality
154:  * is indicated by returning a 0.
155:  */
156: int PROXYCERTINFO_cmp(
157:     const PROXYCERTINFO *                  a,
158:     const PROXYCERTINFO *                  b)
159: {
160:     if(ASN1_INTEGER_cmp(a->path_length, b->path_length) ||
161:         PROXPOLICY_cmp(a->policy, b->policy))
162:     {
163:         return 1;
164:     }
165:     return 0;
166: }
167: /* PROXYCERTINFO_cmp() */
168: /* @} */
169:
170:
171: /**
172:  * @name Print to a BIO stream
173:  */
174: /* @{ */
175: /**
176:  * @ingroup proxycertinfo
177:  *
178:  * print the PROXYCERTINFO structure to stdout
179:  *
180:  * @param bp the BIO to print to
181:  * @param cert_info the PROXYCERTINFO to print
182:  *
183:  * @return 1 on success, 0 on error
184:  */
185: int PROXYCERTINFO_print(
186:     BIO *                                bp,
187:     PROXYCERTINFO *                      cert_info)
188: {
189:     STACK_OF(CONF_VALUE) *                values = NULL;
190:
191:     values = i2v_PROXYCERTINFO(PROXYCERTINFO_x509v3_ext_meth(),
192:                               cert_info,
193:                               NULL);
194:
```

```
195:     X509V3_EXT_val_prn(bp, values, 0, 1);
196:
197:     sk_CONF_VALUE_pop_free(values, X509V3_conf_free);
198:     return 1;
199: }
200: /* PROXYCERTINFO_print() */
201: /* @} */
202:
203:
204: /**
205:  * @name Print To Stream
206:  */
207: /* @{
208: /**
209:  * @ingroup proxycertinfo
210: *
211: * print the PROXYCERTINFO structure to the
212: * specified file stream
213: *
214: * @param fp the file stream (FILE *) to print to
215: * @param cert_info the PROXYCERTINFO structure to print
216: *
217: * @return the number of characters printed
218: */
219: int PROXYCERTINFO_print_fp(
220:     FILE *                                fp,
221:     PROXYCERTINFO *                         cert_info)
222: {
223:     int                                     ret;
224:     BIO *                                    bp;
225:
226:     bp = BIO_new(BIO_s_file());
227:
228:     BIO_set_fp(bp, fp, BIO_NOCLOSE);
229:     ret = PROXYCERTINFO_print(bp, cert_info);
230:     BIO_free(bp);
231:
232:     return (ret);
233: }
234: /* PROXYCERTINFO_print_fp() */
235: /* @} */
236:
237: /**
238:  * @name Set the Policy Field
239:  */
240: /* @{
241: /**
242:  * @ingroup proxycertinfo
243: *
244: * Sets the policy on the PROXYCERTINFO
245: * Since this is an optional field in the
246: * ASN1 encoding, this variable can be set
247: * to NULL through this function - which
248: * means that when the PROXYCERTINFO is encoded
249: * the policy won't be included.
250: *
251: * @param cert_info the PROXYCERTINFO object
252: * to set the policy of
253: * @param policy the PROXPOLICY
254: * to set it to
255: *
256: * @return 1 if success, 0 if error
257: */
258: int PROXYCERTINFO_set_policy(
259:     PROXYCERTINFO *                         cert_info,
```

```
260:     PROXPOLICY *                                policy)
261: {
262:     PROXPOLICY_free(cert_info->policy);
263:     if(policy != NULL)
264:     {
265:         cert_info->policy = PROXPOLICY_dup(policy);
266:     }
267:     else
268:     {
269:         cert_info->policy = NULL;
270:     }
271:     return 1;
272: }
273: /* PROXYCERTINFO_set_policy() */
274: /* @} */
275:
276: /**
277:  * @name Get the Policy Field
278:  */
279: /* @{
280: /**
281:  * @ingroup proxycertinfo
282:  *
283:  * Gets the policy on the PROXYCERTINFO
284:  *
285:  * @param cert_info the PROXYCERTINFO to get the policy of
286:  *
287:  * @return the PROXPOLICY of the PROXYCERTINFO
288:  */
289: PROXPOLICY * PROXYCERTINFO_get_policy(
290:     PROXYCERTINFO *                      cert_info)
291: {
292:     if(cert_info)
293:     {
294:         return cert_info->policy;
295:     }
296:     return NULL;
297: }
298: /* PROXYCERTINFO_get_policy() */
299: /* @} */
300:
301:
302: /**
303:  * @name Set the Path Length Field
304:  */
305: /* @{
306: /**
307:  * @ingroup proxycertinfo
308:  *
309:  * Sets the path length of the PROXYCERTINFO. The path length specifies
310:  * the maximum depth of the path of the Proxy Certificates that
311:  * can be signed by an End Entity Certificate (EEC) or Proxy Certificate.
312:  *
313:  * Since this is an optional field in its ASN1 coded representation,
314:  * it can be set to NULL through this function - which means
315:  * that it won't be included in the encoding.
316:  *
317:  * @param cert_info the PROXYCERTINFO to set the path length of
318:  * @param path_length the path length to set it to
319:  *          if -1 is passed in, the path length gets unset,
320:  *          which configures the PROXYCERTINFO
321:  *          to not include the path length in the DER encoding
322:  *
323:  * @return 1 on success, 0 on error
324:  */
```

```
325: int PROXYCERTINFO_set_path_length(
326:     PROXYCERTINFO *                      cert_info,
327:     long                                path_length)
328: {
329:     if(cert_info != NULL)
330:     {
331:         if(path_length != -1)
332:         {
333:             if(cert_info->path_length == NULL)
334:             {
335:                 cert_info->path_length = ASN1_INTEGER_new();
336:             }
337:             return ASN1_INTEGER_set(cert_info->path_length, path_length);
338:         }
339:         else
340:         {
341:             if(cert_info->path_length != NULL)
342:             {
343:                 ASN1_INTEGER_free(cert_info->path_length);
344:                 cert_info->path_length = NULL;
345:             }
346:             return 1;
347:         }
348:     }
349:     return 0;
350: }
351: /* PROXYCERTINFO_set_path_length() */
352: /* @} */
353:
354:
355: /**
356:  * @name Get Path Length Field
357:  */
358: /* @{ */
359: /**
360:  * @ingroup proxycertinfo
361:  *
362:  * Gets the path length of the PROXYCERTINFO.
363:  *
364:  * @see PROXYCERTINFO_set_path_length
365:  *
366:  * @param cert_info the PROXYCERTINFO to get the path length from
367:  *
368:  * @return the path length of the PROXYCERTINFO, or -1 if not set
369:  */
370: long PROXYCERTINFO_get_path_length(
371:     PROXYCERTINFO *                      cert_info)
372: {
373:     if(cert_info && cert_info->path_length)
374:     {
375:         return ASN1_INTEGER_get(cert_info->path_length);
376:     }
377:     else
378:     {
379:         return -1;
380:     }
381: }
382: /* PROXYCERTINFO_get_path_length() */
383: /* @} */
384:
385:
386: /**
387:  * @name Convert PROXYCERTINFO to DER encoded form
388:  */
389: /* @{ */
```

```
390: /**
391:  * @ingroup proxycertinfo
392:  *
393:  * Converts the PROXYCERTINFO structure from internal
394:  * format to a DER encoded ASN.1 string
395:  *
396:  * @param cert_info the PROXYCERTINFO structure to convert
397:  * @param pp the resulting DER encoded string
398:  *
399:  * @return the length of the DER encoded string
400: */
401: int i2d_PROXYCERTINFO(
402:     PROXYCERTINFO * cert_info,
403:     unsigned char ** pp)
404: {
405:     M_ASN1_I2D_vars(cert_info);
406:
407:     if(cert_info->path_length)
408:     {
409:         M_ASN1_I2D_len(cert_info->path_length, i2d_ASN1_INTEGER);
410:     }
411:
412:     M_ASN1_I2D_len(cert_info->policy, i2d_PROXYPOLICY);
413:
414:     M_ASN1_I2D_seq_total();
415:     if(cert_info->path_length)
416:     {
417:         M_ASN1_I2D_put(cert_info->path_length, i2d_ASN1_INTEGER);
418:     }
419:     M_ASN1_I2D_put(cert_info->policy, i2d_PROXYPOLICY);
420:     M_ASN1_I2D_finish();
421: }
422: /* i2d_PROXYCERTINFO() */
423: /* @} */
424:
425: /**
426:  * @name Convert a PROXYCERTINFO to internal form
427:  */
428: /* @{
429: /**
430:  * @ingroup
431:  *
432:  * Convert from a DER encoded ASN.1 string of a PROXYCERTINFO
433:  * to its internal structure
434: *
435:  * @param cert_info the resulting PROXYCERTINFO in internal form
436:  * @param buffer the DER encoded ASN.1 string containing
437:  * the PROXYCERTINFO
438:  * @param the length of the buffer
439: *
440:  * @return the resultingin PROXYCERTINFO in internal form
441: */
442: PROXYCERTINFO * d2i_PROXYCERTINFO(
443:     PROXYCERTINFO * cert_info,
444:     unsigned char ** pp,
445:     long length)
446: {
447:     M_ASN1_D2I_vars(cert_info, PROXYCERTINFO *, PROXYCERTINFO_new);
448:
449:     M_ASN1_D2I_Init();
450:     M_ASN1_D2I_start_sequence();
451:
452:     M_ASN1_D2I_get_EXP_opt(ret->path_length,
453:                            d2i_ASN1_INTEGER,
454:                            1);
```

```
455:
456:     M_ASN1_D2I_get_opt(ret->path_length,
457:                           d2i_ASN1_INTEGER,
458:                           V_ASN1_INTEGER);
459:
460:     M_ASN1_D2I_get(ret->policy,d2i_PROXYPOLICY);
461:
462:     M_ASN1_D2I_Finish(cert_info,
463:                           PROXYCERTINFO_free,
464:                           ASN1_F_D2I_PROXYCERTINFO);
465: }
466: /* d2i_PROXYCERTINFO( ) */
467: /* @} */
468:
469: X509V3_EXT_METHOD * PROXYCERTINFO_x509v3_ext_meth()
470: {
471:     static X509V3_EXT_METHOD proxycertinfo_x509v3_ext_meth =
472:     {
473:         -1,
474:         X509V3_EXT_MULTILINE,
475:         NULL,
476:         (X509V3_EXT_NEW) PROXYCERTINFO_new,
477:         (X509V3_EXT_FREE) PROXYCERTINFO_free,
478:         (X509V3_EXT_D2I) d2i_PROXYCERTINFO,
479:         (X509V3_EXT_I2D) i2d_PROXYCERTINFO,
480:         NULL, NULL,
481:         (X509V3_EXT_I2V) i2v_PROXYCERTINFO,
482:         NULL,
483:         NULL, NULL,
484:         NULL
485:     };
486:     return (&proxycertinfo_x509v3_ext_meth);
487: }
488:
489: /**
490:  * @name Convert old PROXYCERTINFO to DER encoded form
491:  */
492: /* @{ */
493: /**
494:  * @ingroup proxycertinfo
495:  *
496:  * Converts the old PROXYCERTINFO structure from internal
497:  * format to a DER encoded ASN.1 string
498:  *
499:  * @param cert_info the old PROXYCERTINFO structure to convert
500:  * @param pp the resulting DER encoded string
501:  *
502:  * @return the length of the DER encoded string
503:  */
504: int i2d_PROXYCERTINFO_OLD(
505:     PROXYCERTINFO * cert_info,
506:     unsigned char ** pp)
507: {
508:     int v1;
509:
510:     M_ASN1_I2D_vars(cert_info);
511:
512:     v1 = 0;
513:
514:     M_ASN1_I2D_len(cert_info->policy,
515:                     i2d_PROXYPOLICY);
516:     M_ASN1_I2D_len_EXP_opt(cert_info->path_length,
517:                            i2d_ASN1_INTEGER,
518:                            1, v1);
519:
```

```
520:     M ASN1_I2D_seq_total();
521:     M ASN1_I2D_put(cert_info->policy, i2d_PROXYPOLICY);
522:     M ASN1_I2D_put_EXP_opt(cert_info->path_length, i2d ASN1_INTEGER, 1, v1);
523:     M ASN1_I2D_finish();
524: }
525: /* i2d_PROXYCERTINFO_OLD( ) */
526: /* @} */
527:
528: /**
529:  * @name Convert a old PROXYCERTINFO to internal form
530:  */
531: /* @{ */
532: /**
533:  * @ingroup
534:  *
535:  * Convert from a DER encoded ASN.1 string of a old PROXYCERTINFO
536:  * to its internal structure
537:  *
538:  * @param cert_info the resulting old PROXYCERTINFO in internal form
539:  * @param buffer the DER encoded ASN.1 string containing
540:  * the old PROXYCERTINFO
541:  * @param the length of the buffer
542:  *
543:  * @return the resulting old PROXYCERTINFO in internal form
544:  */
545: PROXYCERTINFO * d2i_PROXYCERTINFO_OLD(
546:     PROXYCERTINFO ** cert_info,
547:     unsigned char ** pp,
548:     long length)
549: {
550:     M ASN1_D2I_vars(cert_info, PROXYCERTINFO *, PROXYCERTINFO_new);
551:
552:     M ASN1_D2I_Init();
553:     M ASN1_D2I_start_sequence();
554:
555:     M ASN1_D2I_get(ret->policy,d2i_PROXYPOLICY);
556:
557:     M ASN1_D2I_get_EXP_opt(ret->path_length,
558:                             d2i ASN1_INTEGER,
559:                             1);
560:
561:     M ASN1_D2I_Finish(cert_info,
562:                        PROXYCERTINFO_free,
563:                        ASN1_F_D2I_PROXYCERTINFO);
564: }
565: /* d2i_PROXYCERTINFO( ) */
566: /* @} */
567:
568: X509V3_EXT_METHOD * PROXYCERTINFO_OLD_x509v3_ext_meth()
569: {
570:     static X509V3_EXT_METHOD proxycertinfo_x509v3_ext_meth =
571:     {
572:         -1,
573:         X509V3_EXT_MULTILINE,
574:         NULL,
575:         (X509V3_EXT_NEW) PROXYCERTINFO_new,
576:         (X509V3_EXT_FREE) PROXYCERTINFO_free,
577:         (X509V3_EXT_D2I) d2i_PROXYCERTINFO_OLD,
578:         (X509V3_EXT_I2D) i2d_PROXYCERTINFO_OLD,
579:         NULL, NULL,
580:         (X509V3_EXT_I2V) i2v_PROXYCERTINFO,
581:         NULL,
582:         NULL, NULL,
583:         NULL
584:     };
}
```

```
585:     return (&proxycertinfo_x509v3_ext_meth);
586: }
587:
588: STACK_OF(CONF_VALUE) * i2v_PROXYCERTINFO(
589:     struct v3_ext_method *                         method,
590:     PROXYCERTINFO *                                ext,
591:     STACK_OF(CONF_VALUE) *                          extlist)
592: {
593:     int                                         len = 128;
594:     char                                         tmp_string[128];
595:
596:     if(!ext)
597:     {
598:         extlist = NULL;
599:         return extlist;
600:     }
601:
602:     if(extlist == NULL)
603:     {
604:         extlist = sk_CONF_VALUE_new_null();
605:         if(extlist == NULL)
606:         {
607:             return NULL;
608:         }
609:     }
610:
611:     if(PROXYCERTINFO_get_path_length(ext) > -1)
612:     {
613:         memset(tmp_string, 0, len);
614:         BIO_snprintf(tmp_string, len, " %lu (0x%lx)",
615:                     PROXYCERTINFO_get_path_length(ext),
616:                     PROXYCERTINFO_get_path_length(ext));
617:         X509V3_add_value("Path Length", tmp_string, &extlist);
618:     }
619:
620:     if(PROXYCERTINFO_get_policy(ext))
621:     {
622:         i2v_PROXYPOLICY(PROXYPOLICY_x509v3_ext_meth(),
623:                           PROXYCERTINFO_get_policy(ext),
624:                           extlist);
625:     }
626:
627:
628:     return extlist;
629: }
```

Attachment 9

proxycertinfo.h

```
1: /*
2:  * Portions of this file Copyright 1999-2005 University of Chicago
3:  * Portions of this file Copyright 1999-2005 The University of Southern California
4:  */
5: /*
6:  * This file or a portion of this file is licensed under the
7:  * terms of the Globus Toolkit Public License, found at
8:  * http://www.globus.org/toolkit/download/license.html.
9:  * If you redistribute this file, with or without
10: * modifications, you must include this notice in the file.
11: */
12: #ifndef HEADER_PROXYCERTINFO_H
13: #define HEADER_PROXYCERTINFO_H
14:
15: /**
16:  * @anchor globus_gsi_proxy_ssl_api
17:  * @mainpage Globus GSI Proxy SSL API
18:  *
19:  * The globus_gsi_proxy_ssl library provides the ability
20:  * to create a PROXYCERTINFO extension for inclusion in
21:  * an X509 certificate. The current specification for the
22:  * extension is described in the Internet Draft
23:  * Document: draft-ietf-pkix-proxy-08.txt
24:  *
25:  * The library conforms to the ASN1 implementation in
26:  * the OPENSSL library (0.9.6, formerly SSLeay), and provides
27:  * an interface to convert from a DER encoded PROXYCERTINFO
28:  * to its internal structure and vice-versa.
29: */
30:
31: #include "proxypolicy.h"
32:
33: #include <openssl/asn1.h>
34: #include <openssl/x509.h>
35:
36: #ifndef EXTERN_C_BEGIN
37: #    ifdef __cplusplus
38: #        define EXTERN_C_BEGIN extern "C" {
39: #        define EXTERN_C_END }
40: #    else
41: #        define EXTERN_C_BEGIN
42: #        define EXTERN_C_END
43: #    endif
44: #endif
45:
46: EXTERN_C_BEGIN
47:
48: #include <openssl/x509.h>
49: #include <openssl/x509v3.h>
50: #include <string.h>
51:
52: /**
53:  * @defgroup proxycertinfo ProxyCertInfo
54:  *
55:  * @author Sam Meder
56:  * @author Sam Lang
57:  *
58:  * The proxycertinfo.h file defines a method of
59:  * maintaining information about proxy certificates.
60: */
61:
62: #define PROXYCERTINFO_OLD_OID          "1.3.6.1.4.1.3536.1.222"
63: #define PROXYCERTINFO_OID              "1.3.6.1.5.5.7.1.14"
64: #define PROXYCERTINFO_SN              "PROXYCERTINFO"
```

```
65: #define PROXYCERTINFO_LN          "Proxy Certificate Info Extension"
66: #define PROXYCERTINFO_OLD_SN      "OLD_PROXYCERTINFO"
67: #define PROXYCERTINFO_OLD_LN      "Proxy Certificate Info Extension (o
ld OID)"
68:
69: /**
70:  * Used for error checking
71: */
72: #define ASN1_F_PROXYCERTINFO_NEW    430
73: #define ASN1_F_D2I_PROXYCERTINFO    431
74:
75:
76: /* data structures */
77:
78: /**
79:  * @ingroup proxycertinfo
80:  *
81:  * This typedef maintains information about a proxy
82:  * certificate.
83:  *
84:  * @note NOTE: The API provides functions to manipulate
85:  * the fields of a PROXYCERTINFO. Accessing the fields
86:  * directly is not a good idea.
87:  *
88:  *
89:  * @param path_length an optional field in the ANS.1 DER encoding,
90:  * it specifies the maximum depth of the path of Proxy Certificates
91:  * that can be signed by this End Entity Certificate or Proxy Certificate.
92:  * @param policy a non-optional field in the ANS.1 DER encoding,
93:  * specifies policies on the use of this certificate.
94: */
95: struct PROXYCERTINFO_st
96: {
97:     ASN1_INTEGER *                      path_length;      /* [ OPTIONAL ] */
98:     PROXPOLICY *                         policy;
99: };
100:
101: typedef struct PROXYCERTINFO_st PROXYCERTINFO;
102:
103: DECLARE_STACK_OF(PROXYCERTINFO)
104: DECLARE_ASN1_SET_OF(PROXYCERTINFO)
105:
106: /* macros */
107:
108: #define d2i_PROXYCERTINFO_bio(bp, pci) \
109:     (PROXYCERTINFO *) ASN1_d2i_bio((char *(*)()) PROXYCERTINFO_new, \
110:     (char *(*)()) d2i_PROXYCERTINFO, \
111:     (bp), (unsigned char **) pci)
112:
113: #define i2d_PROXYCERTINFO_bio(bp, pci) \
114:     ASN1_i2d_bio(i2d_PROXYCERTINFO, bp, \
115:     (unsigned char *) pci)
116:
117: /* functions */
118:
119: ASN1_METHOD * PROXYCERTINFO_asn1_meth();
120:
121: PROXYCERTINFO * PROXYCERTINFO_new();
122:
123: void PROXYCERTINFO_free(
124:     PROXYCERTINFO *                     cert_info);
125:
126: PROXYCERTINFO * PROXYCERTINFO_dup(
127:     PROXYCERTINFO *                     cert_info);
128:
```

```
129: int PROXYCERTINFO_cmp(
130:     const PROXYCERTINFO *                      a,
131:     const PROXYCERTINFO *                      b);
132:
133: int PROXYCERTINFO_print(
134:     BIO *                                bp,
135:     PROXYCERTINFO *                      cert_info);
136:
137: int PROXYCERTINFO_print_fp(
138:     FILE *                                fp,
139:     PROXYCERTINFO *                      cert_info);
140:
141: int PROXYCERTINFO_set_policy(
142:     PROXYCERTINFO *                      cert_info,
143:     PROXPOLICY *                         policy);
144:
145: PROXPOLICY * PROXYCERTINFO_get_policy(
146:     PROXYCERTINFO *                      cert_info);
147:
148: int PROXYCERTINFO_set_path_length(
149:     PROXYCERTINFO *                      cert_info,
150:     long                                path_length);
151:
152: long PROXYCERTINFO_get_path_length(
153:     PROXYCERTINFO *                      cert_info);
154:
155: int i2d_PROXYCERTINFO(
156:     PROXYCERTINFO *                      cert_info,
157:     unsigned char **                     a);
158:
159: PROXYCERTINFO * d2i_PROXYCERTINFO(
160:     PROXYCERTINFO **                    cert_info,
161:     unsigned char **                     a,
162:     long                                length);
163:
164: int i2d_PROXYCERTINFO_OLD(
165:     PROXYCERTINFO *                      cert_info,
166:     unsigned char **                     a);
167:
168: PROXYCERTINFO * d2i_PROXYCERTINFO_OLD(
169:     PROXYCERTINFO **                    cert_info,
170:     unsigned char **                     a,
171:     long                                length);
172:
173: X509V3_EXT_METHOD * PROXYCERTINFO_x509v3_ext_meth();
174:
175: X509V3_EXT_METHOD * PROXYCERTINFO_OLD_x509v3_ext_meth();
176:
177: STACK_OF(CONF_VALUE) * i2v_PROXYCERTINFO(
178:     struct v3_ext_method *               method,
179:     PROXYCERTINFO *                      ext,
180:     STACK_OF(CONF_VALUE) *              extlist);
181:
182: EXTERN_C_END
183:
184: #endif /* HEADER_PROXYCERTINFO_H */
185:
```

Attachment 10

`proxypolicy.c`

```
1: /*
2:  * Portions of this file Copyright 1999-2005 University of Chicago
3:  * Portions of this file Copyright 1999-2005 The University of Southern California
4:  */
5: /*
6:  * This file or a portion of this file is licensed under the
7:  * terms of the Globus Toolkit Public License, found at
8:  * http://www.globus.org/toolkit/download/license.html.
9:  * If you redistribute this file, with or without
10: * modifications, you must include this notice in the file.
11: */
12:
13: #include <stdio.h>
14: #include <openssl/err.h>
15: #include <openssl/asn1_mac.h>
16: #include <openssl/objects.h>
17:
18: #include "proxypolicy.h"
19:
20: /**
21:  * @name Get a method for ASN1 conversion
22:  */
23: /* @{ */
24: /**
25:  * @ingroup proxypolicy
26:  *
27:  * Creates an ASN1_METHOD structure, which contains
28:  * pointers to routines that convert any PROXYPOLICY
29:  * structure to its associated ASN1 DER encoded form
30:  * and vice-versa.
31:  *
32:  * @return the ASN1_METHOD object
33:  */
34:
35: ASN1_METHOD * PROXYPOLICY_asn1_meth()
36: {
37:     static ASN1_METHOD proxypolicy_asn1_meth =
38:     {
39:         (int (*)()) i2d_PROXYPOLICY,
40:         (char **(*)()) d2i_PROXYPOLICY,
41:         (char **(*)()) PROXYPOLICY_new,
42:         (void (*)()) PROXYPOLICY_free
43:     };
44:     return (&proxypolicy_asn1_meth);
45: }
46: /* PROXYPOLICY_asn1_meth() */
47: /* @} */
48:
49: /**
50:  * @name New
51:  */
52: /* @{ */
53: /**
54:  * @ingroup proxypolicy
55:  *
56:  * Allocates and initializes a new PROXYPOLICY structure.
57:  *
58:  * @return pointer to the new PROXYPOLICY
59:  */
60: PROXYPOLICY * PROXYPOLICY_new()
61: {
62:     ASN1_CTX                                c;
63:     PROXYPOLICY *                           ret;
64:
```

```
65:     ret = NULL;
66:
67:     M ASN1_New_Malloc(ret, PROXYPOLICY);
68:     ret->policy_language = OBJ_nid2obj(OBJ_sn2nid(IMPERSONATION_PROXY_SN));
69:     ret->policy = NULL;
70:     return (ret);
71:     M ASN1_New_Error(ASN1_F_PROXYPOLICY_NEW);
72: }
73: /* PROXYPOLICY_new( ) */
74: /* @} */
75:
76:
77: /**
78:  * @name Free
79:  */
80: /* @{ */
81: /**
82:  * @ingroup proxypolicy
83:  *
84:  * Frees a PROXYPOLICY
85:  *
86:  * @param policy the proxy policy to free
87:  */
88: void PROXYPOLICY_free(
89:     PROXYPOLICY *                                policy)
90: {
91:     if(policy == NULL) return;
92:     ASN1_OBJECT_free(policy->policy_language);
93:     M ASN1_OCTET_STRING_free(policy->policy);
94:     OPENSSL_free(policy);
95: }
96: /* PROXYPOLICY_free( ) */
97: /* @} */
98:
99:
100: /**
101:  * @name Duplicate
102:  */
103: /* @{ */
104: /**
105:  * @ingroup proxypolicy
106:  *
107:  * Makes a copy of the proxypolicy - this function
108:  * allocates space for a new PROXYPOLICY, so the
109:  * returned PROXYPOLICY must be freed when
110:  * its no longer needed
111:  *
112:  * @param policy the proxy policy to copy
113:  *
114:  * @return the new PROXYPOLICY
115:  */
116: PROXYPOLICY * PROXYPOLICY_dup(
117:     PROXYPOLICY *                                policy)
118: {
119:     return ((PROXYPOLICY *) ASN1_dup((int (*)())i2d_PROXYPOLICY,
120:                                         (char **(*)())d2i_PROXYPOLICY,
121:                                         (char *)policy));
122: }
123: /* PROXYPOLICY_dup( ) */
124: /* @} */
125:
126:
127: /**
128:  * @name Compare
129:  */
```

```
130: /* @@
131: /**
132: * @ingroup proxypolicy
133: *
134: * Compares two PROXPOLICY structs for equality
135: * This function first compares the policy language numeric
136: * id's, if they're equal, it then compares the two policies.
137: *
138: * @return 0 if equal, nonzero if not
139: */
140: int PROXPOLICY_cmp(
141:     const PROXPOLICY *                      a,
142:     const PROXPOLICY *                      b)
143: {
144:
145:     if((a->policy_language->nid != b->policy_language->nid) ||
146:         ASN1_STRING_cmp((ASN1_STRING *)a->policy, (ASN1_STRING *)b->policy))
147:     {
148:         return 1;
149:     }
150:     return 0;
151: }
152: /* @} */
153:
154:
155: /**
156: * @name Print to a BIO stream
157: */
158: /* @@
159: /**
160: * @ingroup proxypolicy
161: *
162: * Prints the PROXPOLICY struct using the BIO stream
163: *
164: * @param bp the BIO stream to print to
165: * @param policy the PROXPOLICY to print
166: *
167: * @return 1 on success, 0 on error
168: */
169: int PROXPOLICY_print(
170:     BIO *                                bp,
171:     PROXPOLICY *                         policy)
172: {
173:     STACK_OF(CONF_VALUE) *                values = NULL;
174:
175:     values = i2v_PROXPOLICY(PROXPOLICY_x509v3_ext_meth(),
176:                             policy,
177:                             values);
178:
179:     X509V3_EXT_val_prn(bp, values, 0, 1);
180:
181:     sk_CONF_VALUE_pop_free(values, X509V3_conf_free);
182:     return 1;
183: }
184: /* @} */
185:
186:
187: /**
188: * @name Print to a File Stream
189: */
190: /* @@
191: /**
192: * @ingroup proxypolicy
193: *
194: * Prints the PROXPOLICY to the file stream FILE*
```

```
195: *
196: * @param fp the FILE* stream to print to
197: * @param policy the PROXPOLICY to print
198: *
199: * @return number of bytes printed, -2 or -1 on error
200: */
201: int PROXPOLICY_print_fp(
202:     FILE *                                fp,
203:     PROXPOLICY *                           policy)
204: {
205:     int                                ret;
206:
207:     BIO * bp = BIO_new(BIO_s_file());
208:     BIO_set_fp(bp, fp, BIO_NOCLOSE);
209:     ret = PROXPOLICY_print(bp, policy);
210:     BIO_free(bp);
211:
212:     return (ret);
213: }
214: /* @} */
215:
216:
217: /**
218: * @name Set the Policy Language Field
219: */
220: /* @{ */
221: /**
222: * @ingroup proxypolicy
223: *
224: * Sets the policy language of the PROXPOLICY
225: *
226: * @param policy the PROXPOLICY to set the policy language of
227: * @param policy_language the policy language to set it to
228: *
229: * @return 1 on success, 0 on error
230: */
231: int PROXPOLICY_set_policy_language(
232:     PROXPOLICY *                           policy,
233:     ASN1_OBJECT *                          policy_language)
234: {
235:     if(policy_language != NULL)
236:     {
237:         ASN1_OBJECT_free(policy->policy_language);
238:         policy->policy_language = OBJ_dup(policy_language);
239:         return 1;
240:     }
241:     return 0;
242: }
243: /* @} */
244:
245: /**
246: * @name Get the Policy Language Field
247: */
248: /* @{ */
249: /**
250: * @ingroup proxypolicy
251: *
252: * Gets the policy language of the PROXPOLICY
253: *
254: * @param policy the proxy policy to get the policy language
255: * of
256: *
257: * @return the policy language as an ASN1_OBJECT
258: */
259: ASN1_OBJECT * PROXPOLICY_get_policy_language()
```

```
260:     PROXYPOLICY *                                policy)
261: {
262:     return policy->policy_language;
263: }
264: /* @} */
265:
266: /**
267:  * @name Set the Policy Field
268:  */
269: /* @{ */
270: /**
271:  * @ingroup proxypolicy
272:  *
273:  * Sets the policy of the PROXYPOLICY
274:  *
275:  * @param proxypolicy the proxy policy to set the policy of
276:  * @param policy the policy to set it to
277:  * @param length the lenght of the policy
278:  *
279:  * @return 1 on success, 0 on error
280:  */
281: int PROXYPOLICY_set_policy(
282:     PROXYPOLICY *                                proxypolicy,
283:     unsigned char *                                policy,
284:     int                                         length)
285: {
286:     if(policy != NULL)
287:     {
288:         unsigned char *                                copy = malloc(length);
289:         memcpy(copy, policy, length);
290:
291:         if(!proxypolicy->policy)
292:         {
293:             proxypolicy->policy = ASN1_OCTET_STRING_new();
294:         }
295:
296:         ASN1_OCTET_STRING_set(proxypolicy->policy, copy, length);
297:
298:     }
299:     else
300:     {
301:         if(proxypolicy->policy)
302:         {
303:             ASN1_OCTET_STRING_free(proxypolicy->policy);
304:         }
305:     }
306:
307:     return 1;
308: }
309: /* @} */
310:
311:
312: /**
313:  * @name Get the Policy Field
314:  */
315: /* @{ */
316: /**
317:  * @ingroup proxypolicy
318:  *
319:  * Gets the policy of a PROXYPOLICY
320:  *
321:  * @param policy the PROXYPOLICY to get the policy of
322:  * @param length the length of the returned policy - this value
323:  *           gets set by this function
324:  *
```

```
325: * @return the policy
326: */
327: unsigned char * PROXPOLICY_get_policy(
328:     PROXPOLICY *                                policy,
329:     int *                                         length)
330: {
331:     if(policy->policy)
332:     {
333:         (*length) = policy->policy->length;
334:         if(*length > 0 && policy->policy->data)
335:         {
336:             unsigned char *                      copy = malloc(*length);
337:             memcpy(copy, policy->policy->data, *length);
338:             return copy;
339:         }
340:     }
341:
342:     return NULL;
343: }
344: /* */
345:
346:
347: /**
348: * @name Convert from Internal to DER encoded form
349: */
350: /* @{
351: /**
352: * @ingroup proxypolicy
353: *
354: * Converts a PROXPOLICY from its internal structure
355: * to a DER encoded form
356: *
357: * @param a the PROXPOLICY to convert
358: * @param pp the buffer to put the DER encoding in
359: *
360: * @return the length of the DER encoding in bytes
361: */
362: int i2d_PROXPOLICY(
363:     PROXPOLICY *                                a,
364:     unsigned char **                           pp)
365: {
366:     M_ASN1_I2D_vars(a);
367:
368:     M_ASN1_I2D_len(a->policy_language, i2d_ASN1_OBJECT);
369:
370:     if(a->policy)
371:     {
372:         M_ASN1_I2D_len(a->policy, i2d_ASN1_OCTET_STRING);
373:     }
374:
375:     M_ASN1_I2D_seq_total();
376:     M_ASN1_I2D_put(a->policy_language, i2d_ASN1_OBJECT);
377:     if(a->policy)
378:     {
379:         M_ASN1_I2D_put(a->policy, i2d_ASN1_OCTET_STRING);
380:     }
381:     M_ASN1_I2D_finish();
382: }
383: /* */
384:
385:
386: /**
387: * @name Convert from DER encoded form to Internal
388: */
389: /* {@
```

```
390: /**
391:  * @ingroup proxypolicy
392:  *
393:  * Converts the PROXYPOLICY from its DER encoded form
394:  * to an internal PROXYPOLICY structure
395:  *
396:  * @param a the PROXYPOLICY struct to set
397:  * @param pp the DER encoding to get the PROXYPOLICY from
398:  * @param length the length of the DER encoding
399:  *
400:  * @return the resulting PROXYPOLICY in its internal structure
401:  * form - this variable has been allocated using _new routines,
402:  * so it needs to be freed once its no longer used
403:  */
404: PROXYPOLICY * d2i_PROXYPOLICY(
405:     PROXYPOLICY **                                a,
406:     unsigned char **                               pp,
407:     long                                         length)
408: {
409:     M_ASN1_D2I_vars(a, PROXYPOLICY *, PROXYPOLICY_new);
410:
411:     M_ASN1_D2I_Init();
412:     M_ASN1_D2I_start_sequence();
413:     M_ASN1_D2I_get(ret->policy_language, d2i_ASN1_OBJECT);
414:
415:     /* need to try getting the policy using
416:      *      a) a call expecting no tags
417:      *      b) a call expecting tags
418:      * one of which should succeed
419:     */
420:
421:     M_ASN1_D2I_get_opt(ret->policy,
422:                         d2i_ASN1_OCTET_STRING,
423:                         V_ASN1_OCTET_STRING);
424:
425:     M_ASN1_D2I_get_IMP_opt(ret->policy,
426:                            d2i_ASN1_OCTET_STRING,
427:                            0,
428:                            V_ASN1_OCTET_STRING);
429:
430:     M_ASN1_D2I_Finish(a,
431:                        PROXYPOLICY_free,
432:                        ASN1_F_D2I_PROXYPOLICY);
433: }
434: /* @} */
435:
436:
437: X509V3_EXT_METHOD * PROXYPOLICY_x509v3_ext_meth()
438: {
439:     static X509V3_EXT_METHOD proxypolicy_x509v3_ext_meth =
440:     {
441:         -1,
442:         X509V3_EXT_MULTILINE,
443:         NULL,
444:         (X509V3_EXT_NEW) PROXYPOLICY_new,
445:         (X509V3_EXT_FREE) PROXYPOLICY_free,
446:         (X509V3_EXT_D2I) d2i_PROXYPOLICY,
447:         (X509V3_EXT_I2D) i2d_PROXYPOLICY,
448:         NULL, NULL,
449:         (X509V3_EXT_I2V) i2v_PROXYPOLICY,
450:         NULL,
451:         NULL, NULL,
452:         NULL
453:     };
454:     return (&proxypolicy_x509v3_ext_meth);
```

```
455: }
456:
457: STACK_OF(CONF_VALUE) * i2v_PROXYPOLICY(
458:     struct v3_ext_method *                      method,
459:     PROXYPOLICY *                            ext,
460:     STACK_OF(CONF_VALUE) *                  extlist)
461: {
462:     char *                                policy = NULL;
463:     char *                                policy_lang[128];
464:     char *                                tmp_string = NULL;
465:     char *                                index = NULL;
466:     int                                    nid;
467:     int                                    policy_length;
468:
469:     X509V3_add_value("Proxy Policy:", NULL, &extlist);
470:
471:     nid = OBJ_obj2nid(PROXYPOLICY_get_policy_language(ext));
472:
473:     if(nid != NID_undef)
474:     {
475:         BIO_snprintf(policy_lang, 128, " %s", OBJ_nid2ln(nid));
476:     }
477:     else
478:     {
479:         policy_lang[0] = ' ';
480:         i2t_ASN1_OBJECT(&policy_lang[1],
481:                         127,
482:                         PROXYPOLICY_get_policy_language(ext));
483:     }
484:
485:     X509V3_add_value("      Policy Language",
486:                       policy_lang,
487:                       &extlist);
488:
489:     policy = PROXYPOLICY_get_policy(ext, &policy_length);
490:
491:     if(!policy)
492:     {
493:         X509V3_add_value("      Policy", " EMPTY", &extlist);
494:     }
495:     else
496:     {
497:         X509V3_add_value("      Policy:", NULL, &extlist);
498:
499:         tmp_string = policy;
500:         while(1)
501:         {
502:             index = strchr(tmp_string, '\n');
503:             if(!index)
504:             {
505:                 int                  length;
506:                 unsigned char *       last_string;
507:                 length = (policy_length - (tmp_string - policy)) + 9;
508:                 last_string = malloc(length);
509:                 BIO_snprintf(last_string, length, "%8s%s", "", tmp_string);
510:                 X509V3_add_value(NULL, last_string, &extlist);
511:                 free(last_string);
512:                 break;
513:             }
514:
515:             *index = '\0';
516:
517:             X509V3_add_value(NULL, tmp_string, &extlist);
518:
519:             tmp_string = index + 1;
```

```
520:         }
521:         free(policy);
522:     }
523: }
524:
525: return extlist;
526: }
```

Attachment 11

`proxypolicy.h`

```
1: /*
2:  * Portions of this file Copyright 1999-2005 University of Chicago
3:  * Portions of this file Copyright 1999-2005 The University of Southern California
4:  */
5: /*
6:  * This file or a portion of this file is licensed under the
7:  * terms of the Globus Toolkit Public License, found at
8:  * http://www.globus.org/toolkit/download/license.html.
9:  * If you redistribute this file, with or without
10: * modifications, you must include this notice in the file.
11: */
12:
13: #ifndef HEADER_PROXYPOLICY_H
14: #define HEADER_PROXYPOLICY_H
15:
16: /**
17:  * @defgroup proxypolicy ProxyPolicy
18:  *
19:  * @author Sam Meder
20:  * @author Sam Lang
21:  *
22:  * The proxypolicy set of data structures
23:  * and functions provides an interface to generating
24:  * a PROXPOLICY structure which is maintained as
25:  * a field in the PROXYCERTINFO structure,
26:  * and ultimately gets written to a DER encoded string.
27:  *
28:  * @see Further Information about proxy policies
29:  * is available in the Internet Draft Document:
30:  *
31:  * draft-ietf-pkix-proxy-01.txt
32:  */
33:
34: #ifndef EXTERN_C_BEGIN
35: #    ifdef __cplusplus
36: #        define EXTERN_C_BEGIN extern "C" {
37: #        define EXTERN_C_END }
38: #    else
39: #        define EXTERN_C_BEGIN
40: #        define EXTERN_C_END
41: #    endif
42: #endif
43:
44: EXTERN_C_BEGIN
45:
46: #include <openssl/x509.h>
47: #include <openssl/x509v3.h>
48: #include <string.h>
49:
50: #define ANY_LANGUAGE_OID          "1.3.6.1.5.5.7.21.0"
51: #define ANY_LANGUAGE_SN          "ANY_LANGUAGE"
52: #define ANY_LANGUAGE_LN          "Any Language"
53:
54: #define IMPERSONATION_PROXY_OID   "1.3.6.1.5.5.7.21.1"
55: #define IMPERSONATION_PROXY_SN   "IMPERSONATION_PROXY"
56: #define IMPERSONATION_PROXY_LN   "GSI impersonation proxy"
57:
58: #define INDEPENDENT_PROXY_OID     "1.3.6.1.5.5.7.21.2"
59: #define INDEPENDENT_PROXY_SN     "INDEPENDENT_PROXY"
60: #define INDEPENDENT_PROXY_LN     "GSI independent proxy"
61:
62: #define LIMITED_PROXY_OID        "1.3.6.1.4.1.3536.1.1.1.9"
63: #define LIMITED_PROXY_SN        "LIMITED_PROXY"
64: #define LIMITED_PROXY_LN        "GSI limited proxy"
```

```
65:
66: /* Used for error handling */
67: #define ASN1_F_PROXYPOLICY_NEW 450
68: #define ASN1_F_D2I_PROXYPOLICY 451
69:
70: /* data structures */
71:
72: /**
73:  * @ingroup proxypolicy
74:  *
75:  * @note NOTE: The API provides functions to manipulate
76:  * the fields of a PROXYPOLICY. Accessing the fields
77:  * directly will not work.
78:  *
79:  * This typedef maintains information about the policies
80:  * that have been placed on a proxy certificate
81:  *
82:  * @param policy_language defines which policy language
83:  * is to be used to define the policies
84:  * @param policy the policy that determines the policies
85:  * on a certificate
86:  */
87: struct PROXYPOLICY_st
88: {
89:     ASN1_OBJECT * policy_language;
90:     ASN1_OCTET_STRING * policy;
91: };
92:
93: typedef struct PROXYPOLICY_st PROXYPOLICY;
94:
95: DECLARE_STACK_OF(PROXYPOLICY)
96: DECLARE_ASN1_SET_OF(PROXYPOLICY)
97:
98: /* functions */
99:
100: ASN1_METHOD * PROXYPOLICY_asn1_meth();
101:
102: PROXYPOLICY * PROXYPOLICY_new();
103:
104: void PROXYPOLICY_free();
105:
106: PROXYPOLICY * PROXYPOLICY_dup(
107:     PROXYPOLICY * policy);
108:
109: int PROXYPOLICY_cmp(
110:     const PROXYPOLICY * a,
111:     const PROXYPOLICY * b);
112:
113: int PROXYPOLICY_print(
114:     BIO * bp,
115:     PROXYPOLICY * policy);
116:
117: int PROXYPOLICY_print_fp(
118:     FILE * fp,
119:     PROXYPOLICY * policy);
120:
121: int PROXYPOLICY_set_policy_language(
122:     PROXYPOLICY * policy,
123:     ASN1_OBJECT * policy_language);
124:
125: ASN1_OBJECT * PROXYPOLICY_get_policy_language(
126:     PROXYPOLICY * policy);
127:
128: int PROXYPOLICY_set_policy(
129:     PROXYPOLICY * proxypolicy,
```

```
130:     unsigned char *                      policy,
131:     int             length);
132:
133: unsigned char * PROXYPOLICY_get_policy(          policy,
134:     PROXYPOLICY *                  length);
135:     int *                         a);
136:
137: int i2d_PROXYPOLICY(          policy,
138:     PROXYPOLICY *                  a,
139:     unsigned char **                length);
140:
141: PROXYPOLICY * d2i_PROXYPOLICY(          policy,
142:     PROXYPOLICY **                a,
143:     unsigned char **                length);
144:     long
145:
146: X509V3_EXT_METHOD * PROXYPOLICY_x509v3_ext_meth();
147:
148: STACK_OF(CONF_VALUE) * i2v_PROXYPOLICY(          method,
149:     struct v3_ext_method *        ext,
150:     PROXYPOLICY *                extlist);
151:     STACK_OF(CONF_VALUE) *
152:
153: EXTERN_C_END
154:
155: #endif /* HEADER_PROXYPOLICY_H */
```

Attachment 12

`cert-expire.pl`

```
1: #!/usr/bin/perl -w
2:
3: # Usage: cert-expire.pl <cert-file>
4: #
5: # Extract the "Not After" time from an x509 certificate, and then
6: # change the system time to something shortly after that time to
7: # effectively "expire" the certificate.
8: # This is primarily useful for testing cert validity checking.
9:
10: use Time::Local;
11: use Time::gmtime;
12:
13: my $certfile = shift;
14: die "usage: $0 <certfile>" if !defined($certfile);
15: open(CERT, "openssl x509 -in $certfile -noout -enddate|")
16: or die "Can't open $certfile";
17:
18: # Example datetime value: notAfter=Sep 9 16:21:19 2013 GMT
19: my ($monStr,$mday,$hour,$min,$sec,$year,$tz);
20: while (<CERT>) {
21:   next if ! /notAfter/;
22:   print;
23:   ($monStr,$mday,$hour,$min,$sec,$year,$tz) =
24:     $/_ =~ m/notAfter=(\w+)\s+(\w+)\s+(\d+):(\d+):(\d+)\s+(\d+)\s+(\w+)/;
25:   last;
26: }
27: my %mon = ( "Jan", 0,
28:              "Feb", 1,
29:              "Mar", 2,
30:              "Apr", 3,
31:              "May", 4,
32:              "Jun", 5,
33:              "Jul", 6,
34:              "Aug", 7,
35:              "Sep", 8,
36:              "Oct", 9,
37:              "Nov", 10,
38:              "Dec", 11 );
39:
40: $mon = $mon{$monStr};
41: #print "$year $mon $mday $hour $min $sec\n";
42:
43: # now convert the list of time fields into a time value,
44: # increment that value by one hour, and set the system
45: # clock to that datetime.
46:
47: my $time = timelm($sec,$min,$hour,$mday,$mon,$year);
48: my $newTime = $time + 3600;
49: my $tm = gmtime($newTime);
50: $sec = $tm->sec;
51: $min = $tm->min;
52: $hour = $tm->hour;
53: $mday = $tm->mday;
54: $mon = $tm->mon + 1;
55: $year = $tm->year + 1900;
56: #print "$year $mon $mday $hour $min $sec\n";
57:
58: my $dateStr = sprintf("%02d%02d%02d%02d%04d%.02d",
59:                       $mon,$mday,$hour,$min,$year,$sec);
60: print "$dateStr\n";
61: # NOTE: must have 'sudo' permission to change date
62: system("sudo date -u $dateStr");
63:
```

Attachment 13

cert-not-yet-valid.pl

```
1: #!/usr/bin/perl -w
2:
3: # Usage: cert-not-yet-valid.pl <cert-file>
4: #
5: # Extract the "Not Before" time from an x509 certificate, and then
6: # change the system time to something shortly before that time to
7: # effectively make the certificate not-yet-valid.
8: # This is primarily useful for testing cert validity checking.
9:
10: use Time::Local;
11: use Time::gmtime;
12:
13: my $certfile = shift;
14: die "usage: $0 <certfile>" if !defined($certfile);
15: open(CERT, "openssl x509 -in $certfile -noout -startdate|")
16: or die "Can't open $certfile";
17:
18: # Example datetime value: notBefore=Sep 9 16:21:19 2003 GMT
19: my ($monStr,$mday,$hour,$min,$sec,$year,$tz);
20: while (<CERT>) {
21:   next if ! /notBefore/;
22:   print;
23:   ($monStr,$mday,$hour,$min,$sec,$year,$tz) =
24:     $_ =~ m/notBefore=(\w+)\s+(\w+)\s+(\d+):(\d+):(\d+)\s+(\w+)/;
25:   last;
26: }
27: my %mon = ( "Jan", 0,
28:             "Feb", 1,
29:             "Mar", 2,
30:             "Apr", 3,
31:             "May", 4,
32:             "Jun", 5,
33:             "Jul", 6,
34:             "Aug", 7,
35:             "Sep", 8,
36:             "Oct", 9,
37:             "Nov", 10,
38:             "Dec", 11 );
39:
40: $mon = $mon{$monStr};
41: #print "$year $mon $mday $hour $min $sec\n";
42:
43: # now convert the list of time fields into a time value,
44: # decrement that value by one hour, and set the system
45: # clock to that datetime.
46:
47: my $time = timelm($sec,$min,$hour,$mday,$mon,$year);
48: my $newTime = $time - 3600;
49: my $tm = gmtime($newTime);
50: $sec = $tm->sec;
51: $min = $tm->min;
52: $hour = $tm->hour;
53: $mday = $tm->mday;
54: $mon = $tm->mon + 1;
55: $year = $tm->year + 1900;
56: #print "$year $mon $mday $hour $min $sec\n";
57:
58: my $dateStr = sprintf("%02d%02d%02d%02d%04d.%02d",
59:                       $mon,$mday,$hour,$min,$year,$sec);
60: print "$dateStr\n";
61: # NOTE: must have 'sudo' permission to change date
62: system("sudo date -u $dateStr");
63:
```