HP Open Source Security for OpenVMS Volume 2: HP SSL for OpenVMS

HP SSL Version 1.1 for OpenVMS

OpenVMS Alpha Version 7.2-2 or higher, or OpenVMS VAX Version 7.3

This manual supersedes *Open Source Security for OpenVMS Alpha Compaq SSL for OpenVMS Alpha*, Version 7.3-1

Manufacturing Part Number: AA-RSCVB-TE

September 2003

© Copyright 2003 Hewlett-Packard Development Company, L.P.

Legal Notice

Windows®, Windows NT®, and MS Windows® are U.S. registered trademarks of Microsoft Corporation.

UNIX® is a registered trademark of The Open Group in the U.S. and/or other countries.

All other product names mentioned herein may be trademarks of their respective companies.

The information contained herein is subject to change without notice. The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein.

Proprietary computer software. Valid license from HP required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

See Appendix B Open Source Notices for information regarding certain open source code included in this product.

The HP OpenVMS documentation set is available on CD-ROM.

ZK6661

Contents

1. Installation and Release Notes

Installation Requirements and Prerequisites . . . 15
Hardware Prerequisites . . . 15
Software Prerequisites . . . 15
Account Quotas and System Parameters . . . 15
New Features in HP SSL Version 1.1 for OpenVMS . . . 15
OpenSSL Documentation from The Open Group . . . 16
Installing SSL for OpenVMS . . . 16
Postinstallation Tasks . . . 18
SSL Directory Structure . . . 19
Building an SSL Application . . . 19
Building an Application Using 64-Bit APIs . . . 19
Building an Application Using 32-Bit APIs . . . 19
Release Notes . . . 20
Legal Caution . . . 20
Shareable Images Containing 64-Bit and 32-Bit APIs Provided . . . 20
Linking with HP SSL Shareable Images . . . 20
Certificate Verification . . . 20
Preserve Certificates, Keys, and Configuration Files When Upgrading . . . 20
Startup and Shutdown Command Procedure Template Files . . . 21
SSL APIs Not Backward Compatible . . . 21
Certificate Tool Cannot Have Simultaneous Users . . . 21
Protect Certificates and Keys . . . 21
SSL$EXAMPLES Logical Name . . . 21
DES_CBC_CKSUM Return Value Changed to Match Kerberos . . . 21
DES Image Included in SSL V1.1 . . . 22
Environment Variables . . . 23
Known Problem in Multithreaded, 64-bit Applications . . . 23
BIND Error in TCP/IP Application . . . 24
IDEA and RC5 Symmetric Cipher Algorithms Not Supported . . . 24
APIs RAND_egd, RAND_egd_bytes, and RAND_query_egd_bytes Not Supported . . . 24
Compaq C++ V5.5 CANTCOMPLETE Warnings . . . 24
Documentation from the OpenSSL Website . . . 25
Use Certificate Tool for Certificate and Key Creation . . . 25
nsCertType No Longer Written in Certificates . . . 25
Extra Certificate Files — *PEM . . . 25
INDEX.TXT and SERIAL.TXT Location . . . 25

2. Overview of SSL

The SSL Protocol . . . 27
The SSL Handshake . . . 28
Public Key Encryption . . . 29
Certificates . . . 29
Cipher Suite . . . 30
Digital Signatures . . . 30

Contents

3. Using the Certificate Tool

Starting the Certificate Tool . . . 33
Viewing a Certificate . . . 34
View a Certificate Request File . . . 35
Create a Certificate Signing Request . . . 36
  Installing Certificates . . . 38
Create a Self-Signed Certificate . . . 38
Create a Certificate Authority . . . 39
Create a Certificate Chain . . . 41
  Creating an Intermediate CA (RA) Certificate . . . 41
  Creating a Client/Server Certificate Signed with an Intermediate CA Certificate . . . 41
  Creating a Certificate Chain File . . . 41
Sign a Certificate Signing Request . . . 42
Revoke a Certificate . . . 43
Create a Certificate Revocation List . . . 43
Hash Certificates . . . 43
Hash Certificate Revocations . . . 44

4. SSL Programming Concepts

SSL Data Structures . . . 45
  SSL_CTX Structure . . . 46
  SSL Structure . . . 46
  SSL_METHOD Structure . . . 47
  SSL_CIPHER Structure . . . 47
  CERT/X509 Structure . . . 47
  BIO Structure . . . 48
Certificates for SSL Applications . . . 48
  Configuring Certificates in the SSL Client and Server . . . 48
  Obtaining and Creating Certificates . . . 51
SSL Programming Tutorial . . . 52
  Initializing the SSL Library . . . 53
  Creating and Setting Up the SSL Context Structure (SSL_CTX) . . . 54
  Setting Up the Certificate and Key . . . 55
  Creating and Setting Up the SSL Structure . . . 58
  Setting Up the TCP/IP Connection . . . 58
  Setting Up the Socket/Socket BIO in the SSL Structure . . . 59
  SSL Handshake . . . 60
  Transmitting SSL Data . . . 61
  Closing an SSL Connection . . . 61
  Resuming an SSL Connection . . . 62
  Renegotiating the SSL Handshake . . . 63
  Finishing the SSL Application . . . 64

5. OpenSSL Command Line Interface

Command-Line Help . . . 65
Standard Commands . . . 66

Message Digest Commands . . . 68
Encoding and Cipher Commands . . . 68
Password Arguments . . . 71
Creating a DH Parameter (Key) File and a DSA Certificate and Key . . . 71

6. Sample Programs
Programs Included in HP SSL Kit . . . 73
Simple SSL Client Program . . . 74
Simple SSL Server Program . . . 79
Creating Certificates and Keys for the Example Programs . . . 85

CRYPTO and SSL Application Programming Interface (API) Reference . . . 89

A. Data Structures and Header Files
Header Files . . . 597
SSL_CTX Structure . . . 597
SSL Structure. . . 599
SSL_METHOD Structure . . . 602
SSL_SESSION Structure . . . 603
SSL_CIPHER Structure . . . 604
BIO Structure. . . 605
X509 Structure . . . 606

B. Open Source Notices
OpenSSL Open Source License . . . 607
Original SSLeay License . . . 608

Index . . . 609

Tables

Table 4-1. APIs for Data Structure Creation and Deallocation 45
Table 4-2. Types of APIs for SSL_METHOD Creation 54
Table 6-1. HP SSL Example Programs 73
Table 1. HP SSL APIs Grouped by Function 90

Figures

Figure 3-1. Certificate Tool Main Menu 33
Figure 4-1. Relationship Between SSL_CTX and SSL 46
Figure 4-2. Structures Associated with SSL Structure. 47
Figure 4-3. Client and Server Certificates Directly Signed by CAs 48
Figure 4-4. Client and Server Certificates Indirectly Signed by CAs 49
Figure 4-5. Certificates on SSL Client and Server (Case 1) 50
Figure 4-6. Certificates on SSL Client and Server (Case 2) 50
Figure 4-7. Certificate Creation Process 51
Figure 4-8. Overview of SSL Application with OpenSSL APIs 53

Preface

The *HP Open Source Security for OpenVMS, Volume 2: HP SSL for OpenVMS* manual describes how customers can take advantage of the OpenSSL security capabilities available in OpenVMS Alpha and OpenVMS VAX.

Intended Audience

This document is for application developers who want to protect communication links to OpenVMS applications. The OpenSSL APIs establish private, authenticated and reliable communications link between applications.

Document Structure

The information in this manual applies to both OpenVMS Alpha and OpenVMS VAX.

This manual consists of the following chapters:

Chapter 1 contains installation instructions and release notes.

Chapter 2 provides an overview of SSL.

Chapter 3 includes information about the Certificate Tool.

Chapter 4 is a programming tutorial about how to use the OpenSSL APIs in your application program.

Chapter 5 describes the OpenSSL command line interface.

Chapter 6 lists the example programs included in the HP SSL kit.

The CRYPTO and SSL Application Programming Interface (API) Reference is a reference section that includes documentation from The Open Group about the OpenSSL application programming interfaces (APIs).

Appendix A lists the header files and the data structures included in HP SSL for OpenVMS.

Appendix B lists open source notices.

Related Documents

The following documents are recommended for further information:

- *HP Open Source Security for OpenVMS, Volume 1: Common Data Security Architecture*
- *HP Open Source Security for OpenVMS, Volume 3: Kerberos*
- OpenSSL documentation from The Open Group is available at the following World Wide Web address:

  `http://www.openssl.org`

For additional information about HP OpenVMS products and services, see the following World Wide Web address:

`http://www.hp.com/go/openvms/`

For additional information about HP SSL for OpenVMS, see the HP SSL web site at the following World Wide Web address:

`http://h71000.www7.hp.com/openvms/products/ssl/`

Reader's Comments

HP welcomes your comments on this manual.

Please send comments to either of the following addresses:

Internet: openvmsdoc@hp.com

Postal Mail:
Hewlett-Packard Company
OSSG Documentation Group
ZKO3-4/U08
110 Spit Brook Road
Nashua, NH 03062-2698

How to Order Additional Documentation

For information about how to order additional documentation, visit the following World Wide Web address :

`http://www.hp.com/go/openvms/doc/order/`

Conventions

The following conventions may be used in this manual:

Convention	Meaning
Ctrl/x	A sequence such as Ctrl/x indicates that you must hold down the key labeled Ctrl while you press another key or a pointing device button.
PF1 x	A sequence such as PF1 *x* indicates that you must first press and release the key labeled PF1 and then press and release another key (x) or a pointing device button.
Return	In examples, a key name in bold indicates that you press that key.
…	A horizontal ellipsis in examples indicates one of the following possibilities: – Additional optional arguments in a statement have been omitted. – The preceding item or items can be repeated one or more times. – Additional parameters, values, or other information can be entered.
. . .	A vertical ellipsis indicates the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.
( )	In command format descriptions, parentheses indicate that you must enclose choices in parentheses if you specify more than one.
[ ]	In command format descriptions, brackets indicate optional choices. You can choose one or more items or no items. Do not type the brackets on the command line. However, you must include the brackets in the syntax for OpenVMS directory specifications and for a substring specification in an assignment statement.

<table>
<tr><th>Convention</th><th>Meaning</th></tr>
<tr><td>|</td><td>In command format descriptions, vertical bars separate choices within brackets or braces. Within brackets, the choices are optional; within braces, at least one choice is required. Do not type the vertical bars on the command line.</td></tr>
<tr><td>{ }</td><td>In command format descriptions, braces indicate required choices; you must choose at least one of the items listed. Do not type the braces on the command line.</td></tr>
<tr><td>bold type</td><td>Bold type represents the introduction of a new term. It also represents the name of an argument, an attribute, or a reason.

In command or script examples, bold text indicates user input.</td></tr>
<tr><td>italic type</td><td>Italic type indicates important information, complete titles of manuals, or variables. Variables include information that varies in system output (Internal error number), in command lines (/PRODUCER=name), and in command parameters in text (where (dd) represents the predefined par code for the device type).</td></tr>
<tr><td>UPPERCASE TYPE</td><td>Uppercase type indicates a command, the name of a routine, the name of a file, or the abbreviation for a system privilege.</td></tr>
<tr><td><code>Example</code></td><td>This typeface indicates code examples, command examples, and interactive screen displays. In text, this type also identifies URLs, UNIX command and pathnames, PC-based commands and folders, and certain elements of the C programming language.</td></tr>
<tr><td>–</td><td>A hyphen at the end of a command format description, command line, or code line indicates that the command or statement continues on the following line.</td></tr>
<tr><td>numbers</td><td>All numbers in text are assumed to be decimal unless otherwise noted. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated.</td></tr>
</table>

1 Installation and Release Notes

This chapter contains hardware and software prerequisites, installation instructions, postinstallation tasks, instructions for building your application, the SSL directory structure, and release notes for HP SSL V1.1 for OpenVMS. For an overview of SSL, see Chapter 2.

The information in this chapter applies to both OpenVMS Alpha and OpenVMS VAX.

Installation Requirements and Prerequisites

The following sections list hardware and disk space requirements, and software prerequisites.

Hardware Prerequisites

Disk Space Requirements

The SSL for OpenVMS kit requires approximately 45,000 blocks of working disk space to install. Once installed, the software occupies approximately 40,000 blocks of disk space.

Software Prerequisites

SSL for OpenVMS requires the following software:

- OpenVMS Alpha Version 7.2-2 or higher, or OpenVMS VAX Version 7.3
- HP TCP/IP Services for OpenVMS Version 5.3 or higher

SSL for OpenVMS has been tested and verified using HP TCP/IP Services for OpenVMS. There are no known problems running SSL for OpenVMS with other TCP/IP network products. This includes the following TCP/IP network products from Process Software Corporation, but HP has not formally tested and verified these other products:

- TCPware Version 5.5
- MultiNet Version 4.3

Account Quotas and System Parameters

There are no specific requirements for account quotas and system parameters for installing or using SSL for OpenVMS.

New Features in HP SSL Version 1.1 for OpenVMS

SSL Version 1.1 for OpenVMS, based on OpenSSL 0.9.6g, is included in OpenVMS Version 7.3-2. The previous version of OpenVMS included Compaq SSL Version 1.0, which was based on OpenSSL 0.9.6b.

New features in SSL Version 1.1 include:

- A port of the OpenSSL 0.9.6g baselevel, which includes fixes to security vulnerabilities reported on February 19, 2003, and March 17 and 19, 2003 at `http://www.openssl.org/news/`
- Certificate Revocation List (CRL) support in the Certificate Tool
- A DES encryption image that allows you to enable uuencoding and uudecoding
- Three new CRYPTO APIs have been added — BN_pseudo_rand_range, ERR_load_COMP_strings, and X509_STORE_CTX_set_verify_cb
- Two new SSL APIs have been added — SSL_get_rfd and SSL_get_wfd
- One OpenSSL API has been removed — OpenSSLDie

OpenSSL Documentation from The Open Group

Documentation about the OpenSSL project and The Open Group is available at the following URL:

`http://www.openssl.org`

The OpenSSL documentation was written for UNIX® users. When reading UNIX-style OpenSSL documentation, note the following differences between UNIX and OpenVMS:

- File specification format

  The OpenSSL documentation shows example file specifications in UNIX format. For example, the UNIX file specification `/dka100/foo/bar/file.dat` is equivalent to DKA100:[FOO.BAR]FILE.DAT on OpenVMS.
- Directory format

  Directories (pathnames) that begin with a period (.) on UNIX begin with an underscore (_) on OpenVMS. In addition, on UNIX, the tilde (~) is an abbreviation for SYS$LOGIN. For example, the UNIX pathname `~/.openssl/profile/prefs.js` is equivalent to the OpenVMS directory [._OPENSSL.PROFILE]PREFS.JS.

Installing SSL for OpenVMS

SSL for OpenVMS is shipped with OpenVMS   on the Layered Products CD-ROM. You must install SSL before you can use it. Use the following procedure to install SSL for OpenVMS.

To install the SSL for OpenVMS kit, enter the following command:

```
$ PRODUCT INSTALL SSL/SOURCE=ddcu:[dir]
```

By default, SSL for OpenVMS is installed into SYS$SYSDEVICE:[VMS$COMMON]. You can specify a different installation location by using the PRODUCT INSTALL command line qualifier /DESTINATION.

For a description of the features you can request with the PRODUCT INSTALL command when starting an installation, such as running the IVP, purging files, and configuring the installation, refer to the *POLYCENTER Software Installation Utility User's Guide*.

As the installation procedure progresses, the system displays information similar to the following:

```
$ PRODUCT INSTALL SSL

The following product has been selected:
    CPQ AXPVMS SSL V1.1                    Layered Product

Do you want to continue? [YES]

Configuration phase starting ...

You will be asked to choose options, if any, for each selected product and for any products
that may be installed to satisfy software dependency requirements.

CPQ AXPVMS SSL V1.1: SSL for OpenVMS Alpha V1.1 (Based on OpenSSL 0.9.6G).

    (c) Copyright 2003 Hewlett-Packard Development Company, L.P.

Do you want the defaults for all options? [YES]

Do you want to review the options? [NO]

Execution phase starting ...

The following product will be installed to destination:
    CPQ AXPVMS SSL V1.1                    DISK$AXP_X9E9_SSB:[VMS$COMMON.]
The following product will be removed from destination:
    CPQ AXPVMS SSL V1.0-B                  DISK$AXP_X9E9_SSB:[VMS$COMMON.]

Portion done: 0%...10%...20%...30%...40%...50%...60%...70%...80%...90%...100%

The following product has been installed:
    CPQ AXPVMS SSL V1.1                    Layered Product
The following product has been removed:
    CPQ AXPVMS SSL V1.0-B                  Layered Product

%PCSI-I-IVPEXECUTE, executing test procedure for CPQ AXPVMS SSL V1.1 ...

%PCSI-I-IVPSUCCESS, test procedure completed successfully

CPQ AXPVMS SSL V1.1: SSL for OpenVMS Alpha V1.1 (Based on OpenSSL 0.9.6G).

    Insert the following lines in SYS$MANAGER:SYSTARTUP_VMS.COM:
        @sys$startup:ssl$startup.com
    Insert the following lines in SYS$MANAGER:SYSHUTDWN.COM:
        @sys$startup:ssl$shutdown.com

    There are post installation activities that need to be performed.

    This includes things like defining logical names and running SSL$UTILS.COM
    to define some foreign symbols, and running the IVP if it was not done
    as part of the installation.  Refer to the Release Notes for more
    information about activities that should be performed once the installation
    has finished.

    SSL has created the following directory structure in
    PCSI$DESTINATION, which defaults to SYS$SYSDEVICE:[VMS$COMMON]:

     [SSL] -                 Top-level SSL directory
     [SSL.ALPHA_EXE] -       Contains the images for the Alpha platform.
     [SSL.COM] -             Directory to hold the various command procedures.
```

```
     [SSL.DEMOCA] -          Directory structure to demo SSL's CA features
     [SSL.DEMOCA.CERTS] -    Directory to hold the certificates and keys
     [SSL.DEMOCA.CONF] -     Contains the configuration files.
     [SSL.DEMOCA.CRL] -      Contains revoked certificates and CRLs
     [SSL.DEMOCA.PRIVATE] -  Directory for private keys and random data.
     [SSL.DOC] -             OpenSSL Group provided documentation & information.
     [SSL.INCLUDE] -         Contains the C Header (.H) files.
     [SSL.TEST] -            Contains the files used during the IVP.


    Refer to SYS$HELP:SSL011.RELEASE_NOTES for more information.

    @SYS$STARTUP:SSL$STARTUP.COM should be run at system startup.

$
```

Stopping and Restarting the Installation

Use the following procedure to stop and restart the installation:

1. To stop the procedure at any time, press Ctrl/Y.
2. Enter the DCL command PRODUCT REMOVE to reverse any changes to the system that occurred during the partial installation. This deletes all files created up to that point and causes the installation procedure to exit.
3. To restart the installation, go back to the beginning of the installation procedure.

Postinstallation Tasks

After the installation is complete, perform the following steps:

1. Add the following line to the system startup file, SYS$STARTUP:SYSTARTUP_VMS.COM, to set up the SSL symbols and logical names:

   `$ @SYS$STARTUP:SSL$STARTUP`

2. At the DCL command prompt, execute the command that you entered into the system startup file so that you can use SSL immediately:

   `$ @SYS$STARTUP:SSL$STARTUP`

3. Define the foreign commands that use the OpenSSL utility OPENSSL.EXE, such as `openssl, ca, enc, req,` and `X509,` by entering the following command:

   `$ @SSL$COM:SSL$UTILS`

4. Optionally, start the Certificate Tool by entering the following command:

   `$ @SSL$COM:SSL$CERT_TOOL`

   This menu-driven tool allows you to create and view certificates and certificate requests and to sign certifcate requests. For information about the Certificate Tool, see Chapter 3.

SSL Directory Structure

After the installation is complete, the SSL directory structure is as follows:

[SSL] - Top-level directory created by default in SYS$SYSDEVICE:[VMS$COMMON].
[SSL.ALPHA_EXE] - Contains images for the Alpha platform.
[SSL.COM] - Contains command procedures.
[SSL.DEMOCA] - Contains demos for SSL's CA features
[SSL.DEMOCA.CERTS] - Contains certificates and keys.
[SSL.DEMOCA.CONF] - Contains configuration files.
[SSL.DEMOCA.CRL] - Contains revoked certificates and CRLs.
[SSL.DEMOCA.PRIVATE] - Contains private keys and random data.
[SSL.DOC] - OpenSSL Group-provided documentation and information.
[SSL.INCLUDE] - Contains C header (.H) files.
[SSL.TEST] - Contains files used during the Installation Verification Procedure (IVP).

In addition, SSL example programs are located in SYS$COMMON:[SYSHLP.EXAMPLES.SSL]. (The logical name SSL$EXAMPLES points to this directory.) These example programs are also shown and discussed in Chapter 6.

Building an SSL Application

SSL for OpenVMS provides shareable images that contain 64-bit APIs and shareable images that contain 32-bit APIs. You can choose which APIs to use when you compile your application.

The file names for these shareable images are as follows:

SYS$SHARE:SSL$LIBSSL_SHR.EXE - 64-bit SSL APIs
SYS$SHARE:SSL$LIBCRYPTO_SHR.EXE - 64-bit Crypto APIs
SYS$SHARE:SSL$LIBSSL_SHR32.EXE - 32-bit SSL APIs
SYS$SHARE:SSL$LIBCRYPTO_SHR32.EXE - 32-bit Crypto APIs

When you compile your application using HP C, use the /POINTER_SIZE=64 qualifier to take advantage of the 64-bit APIs. The default value for the /POINTER_SIZE qualifier is 32.

Linking your application is the same for both 64-bit or 32-bit APIs. The options file used contains either the 64-bit or 32-bit references to the appropriate shareable image.

Building an Application Using 64-Bit APIs

To build (compile and link) a sample program using the 64-bit APIs, enter the following commands:

```
$ CC/POINTER_SIZE=64/PREFIX=ALL SAMPLE.C
$ LINK/MAP SAMPLE,LINKER_OPT/OPTIONS
```

In these commands, LINKER_OPT.OPT is a simple text file that contains the following lines:

```
SYS$SHARE:SSL$LIBSSL_SHR/SHARE
SYS$SHARE:SSL$LIBCRYPTO_SHR/SHARE
```

Building an Application Using 32-Bit APIs

To build (compile and link) a sample program using the 32-bit APIs, enter the following commands:

```
$ CC/PREFIX=ALL SAMPLE.C
$ LINK/MAP SAMPLE,LINKER_OPT/OPTIONS
```

In these commands, LINKER_OPT.OPT is a simple text file that contains the following lines:

```
SYS$SHARE:SSL$LIBSSL_SHR32/SHARE
SYS$SHARE:SSL$LIBCRYPTO_SHR32/SHARE
```

Release Notes

This section contains notes on the current release of SSL for OpenVMS.

Legal Caution

SSL data transport requires encryption. Many governments, including the United States, have restrictions on the import and export of cryptographic algorithms. Please ensure that your use of SSL is in compliance with all national and international laws that apply to you.

Shareable Images Containing 64-Bit and 32-Bit APIs Provided

SSL for OpenVMS provides shareable images that contain 64-bit APIs and shareable images that contain 32-bit APIs. You can choose which APIs to use when you compile your application. For more information, see Building an SSL Application.

Linking with HP SSL Shareable Images

If you have written an application that links against the OpenSSL object libraries, you must make a minor change to your code because SSL for OpenVMS provides only shareable images. To link your application against the shareable images, use code similar to the following:

```
$ LINK my_app.obj, VMS_SSL_OPTIONS/OPT
```

where VMS_SSL_OPTIONS.OPT is a text file that contains the following lines:

```
SYS$SHARE:SSL$LIBCRYPTO_SHR.EXE/SHARE
SYS$SHARE:SSL$LIBSSL_SHR.EXE/SHARE
```

Certificate Verification

During the SSL handshake, the SSL Protocol verifies the certificates by ensuring that the issue date is between the `notBefore` and `notAfter` dates, and that the trust settings and purpose are valid. The revocation status will not be checked until a version of HP SSL is released that is based on OpenSSL Version 0.9.7, when the proper APIs will be available.

Preserve Certificates, Keys, and Configuration Files When Upgrading

If you are upgrading from a previous version of Compaq SSL to HP SSL V1.1, you must save the certificates, keys, and configuration files in the SSL subdirectory. HP recommends that you back up these items to either a different disk and directory or to tape. When you have completed the V1.1 installation, move the saved items back into the SSL directory structure. Then delete the backed up certificates, keys, and configuration files.

Startup and Shutdown Command Procedure Template Files

In the V1.1 kit, the SYS$STARTUP:SSL$STARTUP.COM and SYS$STARTUP:SSL$SHUTDOWN.COM command procedures are named SYS$STARTUP:SSL$STARTUP.TEMPLATE and SYS$STARTUP:SSL$SHUTDOWN.TEMPLATE. This prevents PCSI from overwriting the .COM files, and allows you to preserve any modifications you made to SSL$STARTUP.COM and SSL$SHUTDOWN.COM if you installed a previous release of SSL for OpenVMS.

After you install the V1.1 kit, compare the new .TEMPLATE files with your existing SSL$STARTUP.COM and SSL$SHUTDOWN.COM files and add any new information as required.

If you did not previously install an SSL for OpenVMS kit, both the .TEMPLATE and .COM files are provided.

Configuration files are provided in the same fashion — both .CNF and .CNF_TEMPLATE files are included in SSL for OpenVMS.

SSL APIs Not Backward Compatible

SSL for OpenVMS is based on open-source code provided by The Open Group. The OpenVMS code is based on the 0.9.6G baselevel of OpenSSL. Until The Open Group releases its Version 1.0 baselevel, The Open Group is not guaranteeing backward compatibility. This means that any OpenSSL API, data structure, header file, command, and the like might be changed in a future version of OpenSSL.

As a result, HP cannot guarantee the backward compatibility of SSL for OpenVMS until the release of SSL for OpenVMS that is based on OpenSSL 1.0.0. The shareable images use EQUAL 1,0 which means that applications will have to relink when new shareable images are distributed.

Certificate Tool Cannot Have Simultaneous Users

Only one user/process should use the Certificate Tool at a time. The tool does not have a locking mechanism to prevent unsynchronized accesses of the database and serial file.

Protect Certificates and Keys

When you create certificates and keys with the Certificate Tool, take care to ensure that the keys are properly protected to allow only the owner of the keys to use them. A private key should be treated like a password. You can use OpenVMS file protections to protect the key file, or you can use ACLs to protect individual key files within a common directory.

SSL$EXAMPLES Logical Name

In SSL V1.1, a new logical, SSL$EXAMPLES, has been added to the SSL$STARTUP.TEMPLATE command procedure. This logical points to the directory SYS$COMMON:[SYSHLP.EXAMPLES.SSL].

DES_CBC_CKSUM Return Value Changed to Match Kerberos

The return value of the DES_CBC_CKSUM API has changed to match its intended compatibility with MIT Kerberos. The DES_CBC_CKSUM routine returns the upper longword of a quadword. The quadword itself was calculated correctly, and has not been changed.

Prior to the change (in Compaq SSL V1.0-B and earlier), the API returned the value in the wrong order. For example:

```
Return value from des_cbc_cksum = 0xaedc29b6
```

In SSL V1.1, the return value is as follows:

```
Return value from des_cbc_cksum = 0xb629dcae
```

This change has been accepted by the OpenSSL.org, and will be available in the 0.9.7A release of OpenSSL.

DES Image Included in SSL V1.1

In the SSL V1.1, an additional image is being made available, called DES.EXE, which is located in the SSL$EXE directory. Create a foreign symbol to access this new image, as follows:

```
$ DES :== $SSL$EXE:DES.EXE
```

The new DES image provides some functionality that is not present in the DES subcommand in the OPENSSL command line interface, most notably the ability to enable uuencoding and uudecoding.

Following is the help text for the DES command and the DES subcommand in the OPENSSL command line interface, which illustrates the differences between the commands.

```
$ DES -?
'?' unknown flag
des <options> [input-file [output-file]]
options:
-v         : des(1) version number
-e         : encrypt using SunOS compatible user key to DES key conversion.
-E         : encrypt
-d         : decrypt using SunOS compatible user key to DES key conversion.
-D         : decrypt
-c[ckname] : generate a cbc_cksum using SunOS compatible user key to
             DES key conversion and output to ckname (stdout default,
             stderr if data being output on stdout).  The checksum is
             generated before encryption and after decryption if used
             in conjunction with -[eEdD].
-C[ckname] : generate a cbc_cksum as for -c but compatible with -[ED].
-k key     : use key 'key'
-h         : the key that is entered will be a hexadecimal number
             that is used directly as the des key
-u[uuname] : input file is uudecoded if -[dD] or output uuencoded data if -[eE]
             (uuname is the filename to put in the uuencode header).
-b         : encrypt using DES in ecb encryption mode, the default is cbc mode.
-3         : encrypt using triple DES encryption.  This uses 2 keys
             generated from the input key.  If the input key is less
             than 8 characters long, this is equivalent to normal
             encryption.  Default is triple cbc, -b makes it triple ecb.

$ OPENSSL DES -?
unknown option '-?'
options are
-in <file>     input file
-out <file>    output file
-pass <arg>    pass phrase source
-e             encrypt
-d             decrypt
-a/-base64     base64 encode/decode, depending on encryption flag
-k             key is the next argument
-kfile         key is the first line of the file argument
-K/-iv         key/iv in hex is the next argument
-[pP]          print the iv/key (then exit if -P)
-bufsize <n>   buffer size
-engine e      use engine e, possibly a hardware device.
Cipher Types
```

```
des     : 56 bit key DES encryption
des_ede :112 bit key ede DES encryption
des_ede3:168 bit key ede DES encryption
rc2     :128 bit key RC2 encryption
bf      :128 bit key Blowfish encryption
 -rc4   :128 bit key RC4 encryption
 -des-ecb      -des-cbc      -des-cfb      -des-ofb      -des  (des-cbc)
 -des-ede      -des-ede-cbc  -des-ede-cfb  -des-ede-ofb  -desx -none
 -des-ede3     -des-ede3-cbc -des-ede3-cfb -des-ede3-ofb -des3 (des-ede3-cbc)
 -rc2-ecb      -rc2-cbc      -rc2-cfb      -rc2-ofb      -rc2  (rc2-cbc)
 -bf-ecb       -bf-cbc       -bf-cfb       -bf-ofb       -bf   (bf-cbc)
 -cast5-ecb    -cast5-cbc   -cast5-cfb    -cast5-ofb    -cast (cast5-cbc)
```

Environment Variables

OpenSSL environmental variables have two formats, as follows:

- $var
- ${var}

In order for these variables to be parsed properly and not be confused with logical names, SSL for OpenVMS only accepts the **${var}** format.

Known Problem in Multithreaded, 64-bit Applications

If you are using HP SSL T1.1 with a multithreaded, 64-bit application, an ACCVIO occurs when the threads are reaped. This problem is under investigation, and there is no known workaround. The ACCVIO looks similar to the following example:

```
%SYSTEM-F-ACCVIO, access violation, reason mask=00, virtual address=0000000000000021,
PC=00000000000A1558, PS=0000001B

  Improperly handled condition, image exit forced.
    Signal arguments:   Number = 0000000000000005
                        Name   = 000000000000000C
                                 0000000000000000
                                 0000000000000021
                                 00000000000A1558
                                 000000000000001B

    Register dump:
    R0  = 0000000080023488  R1  = 000000000000001B  R2  = 0000000000018500
    R3  = 0000000000000021  R4  = 0000000000000003  R5  = 000000000003A1C0
    R6  = 000000000000008C  R7  = 00000000800241C0  R8  = 0000000080023A20
    R9  = 0000000000000000  R10 = 0000000000000001  R11 = 0000000000000001
    R12 = 0000000000000000  R13 = 0000000000000001  R14 = 0000000000000001
    R15 = 0000000000000000  R16 = 0000000080022FE8  R17 = 0000000000000000
    R18 = 0000000000100000  R19 = 000000000000000F  R20 = 0000000000010680
    R21 = 0000000000000003  R22 = 0000000000000001  R23 = 0000000000231B78
    R24 = 00000000800569A0  R25 = 0000000000000003  R26 = 00000000000A1558
    R27 = 000000007BCD03E0  R28 = 0000000000071E20  R29 = 0000000000231B80
    SP  = 0000000000231B80  PC  = 00000000000A1558  PS  = 000000000000001B
%CMA-F-EXIT_THREAD, current thread has been requested to exit
$
```

BIND Error in TCP/IP Application

If you are running a TCP/IP-based SSL client/server application, the server occasionally fails to start up, and displays the following error message:

```
bind: address already in use
```

To avoid this error, use `setsockopt()` with `SO_REUSEADDR` as follows:

```
int   on = 1;
ret = setsockopt(listen_sock, SOL_SOCKET, SO_REUSEADDR, (void *)
&on, sizeof(on));
```

IDEA and RC5 Symmetric Cipher Algorithms Not Supported

The IDEA and RC5 symmetric cipher algorithms are not available in SSL for OpenVMS. Both of these algorithms are under copyright protection, and HP does not have the right to use these algorithms.

If you want to use either of these algorithms, HP recommends that you contact RSA Security at the following URL for the licensing conditions of the RC5 algorithm:

`http://www.rsasecurity.com`

If you want to use the IDEA algorithm, contact Ascom for their license requirements at the following URL:

`http://www.ascom.ch`

Once you have obtained the proper licenses, download the source code from the following URL:

`http://www.openssl.org`

Build the product using the command procedure named MAKEVMS.COM provided in the download.

APIs RAND_egd, RAND_egd_bytes, and RAND_query_egd_bytes Not Supported

The `RAND_egd()`, `RAND_egd_bytes()`, and `RAND_query_egd_bytes()` APIs are not available on OpenVMS.

To obtain a secure random seed on OpenVMS, use the `RAND_poll()` API.

Compaq C++ V5.5 CANTCOMPLETE Warnings

When you compile programs that contain OpenSSL APIs, Compaq C++ Version 5.5 issues warnings about incomplete classes. This error occurs when you use a structure definition before it has been defined. You can resolve these warnings in one of two ways:

- Upgrade to C++ Version 6.0.
- Supply the necessary prototype before using the structure.

The following is an example of this error:

```
$ cxx/list/PREFIX=(ALL_ENTRIES) serv.c
    struct CRYPTO_dynlock_value *data;
........^
%CXX-W-CANTCOMPLETE, In this declaration, the incomplete class
   "unnamed struct::CRYPTO_dynlock_value"
 cannot be completed because it is declared within a
    class or a function prototype.
 at line number 161 in file
    CRYPTO$RES:[OSSL.BUILD_0049_ALPHA_32.INCLUDE.OPENSSL]CRYPTO.H;3
```

Documentation from the OpenSSL Website

The documentation on the OpenSSL website is under development. It is likely that the API and command-line documentation shipped with this kit will differ from the documentation on the OpenSSL website at some point. If such a situation arises, you should consider the API documentation on the OpenSSL website to have precedence over the documentation included in this kit.

Use Certificate Tool for Certificate and Key Creation

HP recommends the use of the Certificate Tool (SSL$COM:SSL$CERT_TOOL.COM) when creating certificates and keys to test your SSL application. The Certificate Tool provides both ease of use and consistency when creating your certificates and keys to test and demonstrate your SSL client and server application.

nsCertType No Longer Written in Certificates

In the SSL T1.0 field test kit, the Certificate Tool incorrectly set the nsCertType field with both server and client values. The field should have been set with one value, either server or client, but not both. In Version 1.0 and higher releases, this field is not set in the Certificate Tool. Your application is still able to pass certificates as either server or client certificates, but object signing cannot be completed with a null nsCertType field.

If object signing is required in your application, see the following paragraphs about setting values in the nsCertType field.

HP recommends that you delete the nsCertType field from the existing SSL$CONF:SSL$CA.CNF file by editing the file and deleting the line that begins with nsCertType =.

If you have an application that requires the nsCertType field, edit the file SSL$CONF:SSL$CA.CNF and enter the value that your application requires. For example, if your application needs a certificate with the client nsCertType field value, enter the following:

```
nsCertType = client
```

Valid values for the nsCertType field are server, client, email, objsign, sslCA, emailCA, and objCA.

Extra Certificate Files — *PEM

When you sign a certificate request using either the Certificate Tool or the OpenSSL utility, you may notice that an extra certificate is produced with a name similar to SSL$CRT01.PEM. This certificate is the same as the certificate that you produced with the name you chose. These extra files are the result of the OpenSSL demonstration Certificate Authority (CA) capability, and are used as a CA accounting function. These extra files are kept by the CA and can be used to generate Certificate Revocation Lists (CRLs) if the certificate becomes compromised.

INDEX.TXT and SERIAL.TXT Location

In the Compaq SSL T1.0 field test kit, INDEX.TXT and SERIAL.TXT were located in SSL$ROOT:[DEMOCA.PRIVATE]. In Compaq SSL Version 1.0 and higher releases, these files are located in SSL$ROOT:[DEMOCA].

The location of INDEX.TXT and SERIAL.TXT is controlled by the OPENSSL-VMS.CNF file, and consumed by the OpenSSL utility and the Certificate Tool as part of the OpenSSL demonstration Certificate Authority (CA) database.

2 Overview of SSL

Secure Sockets Layer (SSL) is the open standard security protocol for the secure transfer of sensitive information over the Internet. SSL provides three things: privacy through encryption, server authentication, and message integrity. Client authentication is available as an optional function.

OpenVMS includes three standards-based cryptographic security solutions, HP SSL for OpenVMS, **Common Data Security Architecture (CDSA)**, and **Kerberos for OpenVMS** that protect your information and communications.

Protecting communication links to OpenVMS applications over a TCP/IP connection can be accomplished through the use of SSL. The OpenSSL APIs establish private, authenticated and reliable communications links between applications.

CDSA for OpenVMS provides a security infrastructure that allows for the creation of multiplatform, open source industry standard cryptographic solutions. CDSA provides a flexible mix-and-match solution among a variety of different applications and security services. This allows for compliance to local regulation while keeping the security underpinnings transparent to the end user. For more information, see the *Open Source Security for OpenVMS, Volume 1: Common Data Security Architecture*.

Kerberos is a network authentication protocol designed to provide strong authentication for client/server applications by using secret-key cryptography. It was developed at the Massachusetts Institute of Technology as part of Project Athena in the mid-1980s. The Kerberos protocol uses strong cryptography, so that a client can prove its identity to a server (and vice versa) across an insecure network connection. After a client and server have used Kerberos to prove their identity, they can also encrpt all of their communications to assure privacy and data integrity. For more information, see *Open Source Security for OpenVMS, Volume 3: Kerberos*.

NOTE SSL data transport requires encryption. Many governments, including the United States, have restrictions on the import and export of cryptographic algorithms. Please ensure that your use of SSL is in compliance with all national and international laws that apply to you.

This chapter discusses the following topics:

- The SSL protocol
- The SSL handshake
- Public key encryption
- Certificates
- Cipher suite
- Digital signatures

The SSL Protocol

This section provides an overview of SSL technology and its application.

The SSL protocol works cooperatively on top of several other protocols. SSL works at the application level. The underlying mechanism is TCP/IP (Transmission Control Protocol/Internet Protocol), which governs the transport and routing of data over the Internet. Application protocols, such as HTTP (HyperText Transport Protocol), LDAP (Lightweight Directory Access Protocol), and IMAP (Internet Messaging Access Protocol), run on top of TCP/IP. They use TCP/IP to support typical application tasks, such as displaying web pages or running email servers.

SSL addresses three fundamental security concerns about communication over the Internet and other TCP/IP networks:

- **SSL server authentication** — Allows a user to confirm a server's identity. SSL-enabled client software can use standard techniques of public-key cryptography to check whether a server's certificate and public ID are valid and have been issued by a Certificate Authority (CA) listed in the client's list of trusted CAs. Server authentication is used, for example, when a PC user is sending a credit card number to make a purchase on the web and wants to check the receiving server's identity.
- **SSL client authentication** — Allows a server to confirm a user's identity. Using the same techniques as those used for server authentication, SSL-enabled server software can check whether a client's certificate and public ID are valid and have been issued by a Certificate Authority (CA) listed in the server's list of trusted CAs. Client authentication is used, for example, when a bank is sending confidential financial information to a customer and wants to check the recipient's identity.
- **An encrypted SSL connection** — Requires all information sent between a client and a server to be encrypted by the sending software and decrypted by the receiving software, thereby providing a high degree of confidentiality. Confidentiality is important for both parties to any private transaction. In addition, all data sent over an encrypted SSL connection is protected with a mechanism that automatically detects whether data has been altered in transit.

The SSL Handshake

An SSL session always begins with an exchange of messages called the **SSL handshake**. The handshake allows the server to authenticate itself to the client using public key techniques, also called asymmetric encryption. It then allows the client and the server to cooperate in the creation of symmetric keys, which are used for rapid encryption, decryption, and tamper detection during the session that follows. Optionally, the handshake also allows the client to authenticate itself to the server.

This exchange of messages is designed to facilitate the following actions:

- Authenticate the server to the client.
- Allow the client and server to select the cryptographic algorithms, or ciphers, that they both support.
- Optionally authenticate the client to the server.
- Use public key encryption techniques to generate shared secrets.
- Establish an encrypted SSL connection.

Public Key Encryption

In traditional environments, encrypted information is sent between parties that use the same key to encode and decode information. This is called **symmetric encryption**. In the case of the Internet, there is no way for one computer to send the encryption key to another without risk of a third party stealing the key and decoding subsequent communications. A method other than symmetrical encryption is required to transmit the encryption key securely on the Internet.

Public key cryptography was developed by Whitfield Diffie and Martin Hellman. The Diffie-Hellman key agreement protocol was published in 1976. It is also called **asymmetric encryption** because it uses two keys instead of one key. The RSA algorithm is another option for public key cryptography.

The solution is a system called **public key cryptography** or **asymmetric encryption**, which uses two keys. One is a **public key** and is usually available to anyone who wants it. The other, a **private key**, is held by just one party. Only the private key can decipher information that is encrypted using the public key; it is impossible to decipher the message using the public key. Similarly, only the private key can create encrypted messages that are decipherable with the public key. Because there can be only one public key for each private key, and vice-versa, it is nearly impossible to impersonate the holder of the private key. The two keys are mathematically related, but in such a way that it is virtually impossible to derive the private key from the public one.

During the SSL handshake, each computer generates a set of codes to encrypt information. From these codes, each computer creates two keys, one private key and one public key. Your computer keeps the private key secret, but it sends out the public key to the other computer, which uses that key to encode subsequent messages that only your computer can read. However, the public key cannot, be used to decode the message; only private key can decode the message.

These keys allow you and the other computer to lock and unlock information so that only the holder of the private key can read messages encrypted by the public key. Since only you and the other computer have a copy of your respective private keys, there is no way for anybody else to intercept and decode your messages.

Certificates

A **certificate**, or digital certificate, is an electronic document used to identify an individual, a server, a company, or some other entity and to associate that identity with a public key. Like a driver's license, a passport, or other commonly used personal IDs, a certificate provides generally recognized proof of a person's identity. Public key cryptography uses certificates to address the problem of impersonation.

Certificates are issued by **certificate authorities**. The Certificate Authority (CA) is a trusted third party that verifies the identity of the site with which you are connected. Like any form of identification, the authenticity of the issuer is essential.

The role of CAs in validating identities and in issuing certificates is analogous to the way a government issues passports and driver's licenses. CAs can be either independent third parties or organizations running their own certificate-issuing server software (such as Netscape Certificate Server).

The methods used to validate an identity vary depending on the policies of a given CA. In general, before issuing a certificate, the CA must use its published verification procedures for that type of certificate to ensure that an entity requesting a certificate is in fact who it claims to be.

The certificate issued by the CA binds a particular public key to the name of the entity the certificate identifies (such as the name of an employee or a server). Certificates help prevent the use of fake public keys for impersonation. Only the public key certified by the certificate works with the corresponding private key possessed by the entity identified by the certificate.

In addition to a public key, a certificate always includes the name of the entity it identifies, an expiration date, the name of the CA that issued the certificate, a serial number, and other information. Most importantly, a certificate always includes the **digital signature** of the issuing CA. The CA's digital signature allows the certificate to function as a "letter of introduction" for users who know and trust the CA but who do not know the entity identified by the certificate.

For information about the HP SSL Certificate Tool, which allows you to view and create certificates, see Chapter 3.

Cipher Suite

Integral to the SSL protocol is its use of cryptographic algorithms, generally called ciphers. **Ciphers** are required to authenticate the server and client to each other, transmit certificates, and establish session keys. Clients and servers can support different cipher suites, or sets of ciphers, depending on factors such as the version of SSL they support, company policies regarding acceptable encryption strength, and government restrictions on the export of SSL-enabled software.

Among its other functions, the SSL handshake protocol determines how the server and client negotiate which cipher suites they will use to authenticate each other, to transmit certificates, and to establish session keys. Key exchange algorithms such as RSA and DH key exchange govern the way the server and client determine the symmetric keys they will both use during an SSL session. The most commonly used SSL cipher suites use RSA key exchange.

The SSL 2.0 and SSL 3.0 protocols support overlapping sets of cipher suites. Administrators can enable or disable any of the supported cipher suites for both clients and servers. When a particular client and server exchange information during the SSL handshake, they identify the strongest enabled cipher suites they have in common and use those for the SSL session.

Decisions about which cipher suites a particular organization decides to enable depend on trade-offs among the sensitivity of the data involved, the speed of the cipher, and the applicability of export rules.

Digital Signatures

Encryption and decryption address the problem of eavesdropping. However, tampering and impersonation are still possible.

Public key cryptography addresses the problem of tampering using a mathematical function called a **one-way hash function** (also called a message digest function or algorithm). A one-way hash is a fixed-length number whose value is unique to the data being hashed. Any change in the data, even deleting or altering a single character, results in a different value.

For all practical purposes, the content of the hashed data cannot be deduced from the hash, which is why it is called "one-way."

This principle is the crucial part of digitally signing any data. Instead of encrypting the data itself, the signing software creates a one-way hash of the data, then uses your private key to encrypt the hash. The encrypted hash, along with other information, such as the hashing algorithm, is known as a **digital signature**.

3 Using the Certificate Tool

HP SSL for OpenVMS provides a certificate tool that is a simple menu-driven interface for viewing and creating SSL certificates. The OpenSSL Certificate Tool enables you to perform the most important certification functions with ease. Using it, you can view certificates and certificate requests, create certificate requests, sign your own certificate, create your own certificate authority, and sign client certificate requests. Additional hash functions are included.

NOTE Some OpenSSL commands are beyond the scope of the Certificate Tool. For these, use the command-line OpenSSL utility. See Chapter 5 for more information

Starting the Certificate Tool

Run the Certificate Tool by entering the following command at the DCL command prompt:

```
$ @SSL$COM:SSL$CERT_TOOL
```

NOTE Only one user/process should use the Certificate Tool at a time. The tool does not have a locking mechanism to prevent unsynchronized accesses of the database and serial file. This assumes that you started SSL using SSL$STARTUP.COM.

Figure 3-1 shows the Certificate Tool's main menu.

Figure 3-1 Certificate Tool Main Menu

```
SSL Certificate Tool

Main Menu

1.  View a Certificate
2.  View a Certificate Signing Request
3.  Create a Certificate Signing Request
4.  Create a Self-Signed Certificate
5.  Create a CA (Certification Authority) Certificate
6.  Sign a Certificate Signing Request
7.  Revoke Certificates
8.  Create a Certificate Revocation List
9.  Hash Certificates
10. Hash Certificates Revocations
11. Exit

Enter Option:
```

VM-0868A-AI

Viewing a Certificate

The content of a certificate associates a public key with the real identity of an individual, server, or other entity (known as the **subject**). Information about the subject includes identifying information (the distinguished name), and the public key. It also includes the identification and signature of the certificate authority that issued the certificate, and the period of time during which the certificate is valid. The certificate might contain additional information (or extensions) as well as administrative information, such as a serial number, for the Certificate Authority's use.

To view a certificate, do the following:

1. Select the View a Certificate option from the main menu by entering 1 and pressing enter.
2. Press enter to accept the default file specification (or type a new file specification to an alternative location) for the certificate directory to find files with a CRT extension:

```
SSL Certificate Tool

View Certificate

Display Certificate File: ? [SSL$CRT:*.CRT]
```

VM-0869A-AI

   The default directory specification of SSL$CRT: is where certificates you sign are saved. Server certificates can be saved on your system by other products. For example, HP Secure Web Server for OpenVMS Alpha places certificates in APACHE$ROOT:[CONF.SSL_CRT].

3. Select a certificate file by entering its number, then pressing Enter. In the following example, number 1 (server_ca.crt) was selected.

```
SSL Certificate Tool

View Certificate

<Select a File>      Page 1 of 1

1. SSL$ROOT:[CERTS]server_ca.crt;1
2. SSL$ROOT:[CERTS]test_selfsign.crt;1
3. SSL$ROOT:[CERTS]TEST_SELFSIGN_X509.CRT;1

Enter B for Back, N for Next, Ctrl-Z to Exit or Enter a File Number
```

VM-0870A-AI

4. View the certificate details:
   - Version (SSL 3.0 protocol)
   - Serial number (Certificates issued by a CA have a serial number that is unique to the certificates issued by that CA.)

- Signature algorithm
- Issuer
- Validity (inception and expiration dates)
- Public key information

This information is displayed as follows:

```
                    SSL Certificate Tool

                      View Certificate

          < SSL$ROOT:[CERTS]server_ca.crt;1 >          Page 1 of 1

Certificate:
  Data:
      Version: 3 (0x2)
       Serial Number: 0 (0x0)
       Signature Algorithm: md5WithRSAEncryption
       Issuer: C=US, O=Compaq Computer Corp., OU=OpenVMS, CN=Dwllng CA Authority
   Validity
         Not Before: Jan 24 02:26:16 2002 GMT
         Not After : Jan 23 02:26:16 2007 GMT
     Subject: C=US, O=Compaq Computer Corp., OU=OpenVMS, CN=Dwllng CA Authority
     Subject Public Key Info:
          Public Key Algorithm: rsaEncryption
          RSA Public Key: (1024 bit)
             Modulus (1024 bit):
                 00:c5:6e:63:90:d7:11:d8:13:a8:96:8a:a3:4f:dd:
                 d3:8b:e6:d7:77:2c:8e:72:e6:63:73:14:1c:a9:be:
                 30:05:8e:84:74:17:cb:56:b3:7b:31:d4:44:26:8f:
                 b4:72:cf:22:f9:96:ea:84:b8:d0:13:0e:e4:cb:08:
                 25:e9:2e:3a:c8:32:06:39:71:ee:93:a4:f4:71:f2:
                 e2:91:35:b8:6e:d3:5a:b2:0c:d9:a0:fe:07:f7:5d:
                 ed:89:77:77:41:3c:0d:bc:6a:41:b6:2e:1c:a6:3c:
                 81:3f:70:3c:58:a3:63:3d:cd:57:2a:d3:28:97:39:
                 f3:dd:33:65:a9:09:21:b6:bb
             Exponent: 65537 (0x10001)
          Signature Algorithm: md5WithRSAEncryption
                5c:ea:12:35:de:24:c7:c0:40:ca:90:57:9b:31:b2:c4:79:fc:
                a6:b2:fa:b4:fe:43:92:94:66:20:01:ec:63:0c:32:57:63:fe:
                92:a7:bb:8c:a1:4f:92:15:6f:75:b7:9a:9d:a8:e6:59:51:77:
                2c:61:99:d3:2c:52:8c:db:d2:b8:a7:21:44:3d:b2:16:22:0b:
                39:97:5b:84:9e:68:30:cb:74:d9:cf:03:c4:95:b0:d7:7a:09:
                45:28:6d:29:eb:83:1f:76:13:6e:78:8d:eb:c5:54:d9:dc:71:
                32:1e:be:2d:a1:d0:67:95:03:8f:bd:c6:0b:f3:54:93:b8:1f:
                b8:96

            ~~~~~~Enter B for Back, N for Next, Ctrl-Z to Exit ~~~~~~
```

VM-0871A-AI

View a Certificate Request File

A certificate request file is an unsigned certificate.

To view a certificate request file, do the following:

1. Type the file specification to the certificate request directory to find files with a .CSR extension:
2. Select a certificate request file.
3. View the certificate request details:
   - Subject
   - Public key information
   - Signature algorithm
   - Issuer
   - Validity (inception and expiration dates)

Create a Certificate Signing Request

Creating a certificate signing request (generating a *.CSR file) is like an application form for a certificate. You can specify two categories of request:

- Server certificate request

  Prepares a certificate file to be signed by a trusted (root) CA to authenticate your server. You are the subject of the certificate, and the CA you send it to will be the certificate issuer. For example, if you wanted to get a Thawte Server ID, you would create a certificate request and mail the contents of this generated file to Thawte. The file you generate is a *.CSR file.

- Client certificate request

  Prepares client certificate files that are loaded in the SSL client application, such as a web browser. The client is the subject of the certificate and you are the certificate issuer.

To create a certificate request, perform the following steps.

1. Enter the information required for the certificate. You must complete all fields to create a valid certificate request. The certificate request is generated after you respond to the last question.
   - Encrypt Private Key

     Using an encrypted private key forces the passphrase dialog when loading the private key.

NOTE Do not use this option if you are using the `mod_ssl` directive `SSLPassPhraseDialog` with the default built-in option.

   - Encryption Bits

     The largest recommended size is 1024 bits. Encryption strength is often described in terms of the size of the keys used to perform the encryption; in general, longer keys provide stronger encryption but require more computing time. Key length is measured in bits. Private key sizes larger than 1024 bits are incompatible with some versions of Netscape Navigator and Microsoft Internet Explorer.

   - Certificate Key File

     Use OpenVMS syntax (defaults to SSL$KEY:SERVER.KEY).

   - Certificate Request File

Use OpenVMS syntax (defaults to SSL$CSR:SERVER.CSR).

The remaining questions determine your server's distinguished name.

- Country Name
- State or Province Name
- City Name
- Organization Name
- Organization Unit Name
- Common Name

  Common name usage is different for client certificates than it is for server certificates. Generally, the common name on a client certificate is the proper name of the individual requesting a certificate. In the case of server certificates, the common name must be the same as your server's DNS host name (or virtual host name, if name-based virtual hosting is used). Browsers compare the common name in the server certificate with the host name of the server to which they are connecting; these names must match.
- Email Address
- Display the Certificate

2. View the details of the certificate request (if you chose to display the certificate).

   - Subject
   - Public key information
   - Signature algorithm

To see the encoded contents, exit the certificate tool and enter the following command to view the CSR file.

```
$ TYPE SSL$ROOT:[CERTS]SERVER.CSR
```

What you see is exactly what is required by the certificate authority. You might be required to send the file itself or just the contents of the file to your CA (according to the CA's instructions). For example:


```
-----BEGIN CERTIFICATE REQUEST-----
MIIB/TCCAWYCAQAwgbwxCzAJBgNVBAYTAlVTMRYwFAYDVQQIEw1OZXcgSGFtcHNo
aXJlMQ8wDQYDVQQHEwZOYXNodWExHjAcBgNVBAoTFUNvbXBhcSBDb21wdXRlciBD
b3JwLjEcMBoGA1UECxMTT3BlblZNUyBFbmdpbmVlcmluZzEaMBgGA1UEAxMRRkxJ
UDMuWktPLkRFQy5DT00xKjAoBgkqhkiG9w0BCQEWG3dlYm1hc3RlckBGTElQMy5a
S08uREVDLkNPTTCBnzANBgkqhkiG9w0BAQEFAAOBjQAwgYkCgYEA0/y8RxuE/COy
nVpeK00GgvbgFWxX1o89ULQTMVUSwmAzhdzbi3DZL5s85YRGdPVgYW2rWs1t2SQg
jMSlFTxta/CwW6Vwwn9GmdaJwkqGFxnpw2LmugexLfj+4t97AZyIR2O7gJxCINS5
CWg3tcn1ZUmqswjkrG8WehUN+2C6IBcCAwEAAaAAMA0GCSqGSIb3DQEBBAUAA4GB
ABzgiiojPAcojLXGI2OFxJ5apORAHHHAyc0YCuhFXS1Rs2BIXHmM5xQuxk8yitc4
yViQfHhGDzpDmOwMKkK7t09UjQh9humKEUlAnS4VYLL4VlgenwLybcLLB0Q3aiQN
UjQw9RrXNWWZYVDenvrOwtbK9dFefb4PlZIAS2/Z4jLP
-----END CERTIFICATE REQUEST-----
```


If you are sending only the contents, copy and paste everything and send to the CA using secure email or the appropriate enrollment form. The CA will return a digitally signed certificate to you. For example:


```
-----BEGIN CERTIFICATE-----
MIICeDCCAiICEEdpjxOzmJPyh5TiG8BRA70wDQYJKoZIhvcNAQEEBQAwgakxFjAU
BgNVBAoTDVZlcmlTaWduLCBJbmMxRzBFBgNVBAsTPnd3dy52ZXJpc2lnbi5jb20v
cmVwb3NpdG9yeS9UZXN0Q1BTIEluY29ycC4gQnkgUmVmLiBMaWFiLiBMVEQuMUYw
RAYDVQQLEz1Gb3IgVmVyaVNpZ24gYXV0aG9yaXplZCB0ZXN0aW5nIG9ubHkuIE5v
IGFzc3VyYW5jZXMgKEMpVlMxOTk3MB4XDTAwMDcwNzAwMDAwMFoXDTAwMDcyMTIz
```

```
NTk1OVowgZAxCzAJBgNVBAYTAlVTMRYwFAYDVQQIEw1OZXcgSGFtcHNoaXJlMQ8w
DQYDVQQHFAZOYXNodWExHjAcBgNVBAoUFUNvbXBhcSBDb21wdXRlciBDb3JwLjEc
MBoGA1UECxQTT3BlblZNUyBFbmdpbmVlcmluZzEaMBgGA1UEAxQRRkxJUDMuWktP
LkRFQy5DT00wgZ8wDQYJKoZIhvcNAQEBBQADgY0AMIGJAoGBANP8vEcbhPwjsp1a
XitNBoL24BVsV9aPPVC0EzFVEsJgM4Xc24tw2S+bPOWERnT1YGFtq1rNbdkkIIzE
pRU8bWvwsFulcMJ/RpnWicJKhhcZ6cNi5roHsS34/uLfewGciEdju4CcQiDUuQlo
N7XJ9WVJqrMI5KxvFnoVDftguiAXAgMBAAEwDQYJKoZIhvcNAQEEBQADQQAySLLe
U7nMLJ+QkRld6iqKjU2VotphPvgWMGsJ+TKqUI4MXaAv0zQxtBni1N8s0LXVNCuJ
lEzBYjSbgbgEhJJA
-----END CERTIFICATE-----
```


The CA-signed certificate contains the following information:

- Your organization's common name (www.*your-server*)
- Additional identifying information (IP and physical address)
- Your public key
- Expiration date of the public key
- Name of the CA that issued the ID
- A unique serial number. (Every certificate issued by a CA has a serial number that is unique to the certificates issued by that CA.)
- CA's digital signature

Installing Certificates

A signed certificate needs to be installed, along with the key you generated when creating the request, by saving or copying the respective files to their correct directories and restarting the application.

The following example shows a certificate and key copied to the directory of a web server.

```
$ COPY SSL$CERTS:SERVER.CRT APACHE$SPECIFIC:[CONF.SSL_CRT]

$ COPY SSL$KEY:SERVER.KEY APACHE$SPECIFIC:[CONF.SSL_KEY]
```

Create a Self-Signed Certificate

To create a self-signed certificate, perform the following steps. All fields must be completed to create a valid self-signed certificate. The inception time of a certificate is based on UTC (Coordinated Universal Time). Check with your system administrator that your computer's UTC is set correctly if you want to use the self-signed certificate right away.

1. Enter the required information for the self-signed certificate.

   - Encrypt Private Key

     Using an encrypted private key forces the passphrase dialog to appear at startup time.

   - Encryption Bits

The largest recommended size is 1024 bits. Encryption strength is often described in terms of the size of the keys used to perform the encryption; in general, longer keys provide stronger encryption. Key length is measured in bits. Private key sizes larger than 1024 bits are incompatible with some versions of Netscape Navigator and Microsoft Internet Explorer.

- Certificate Key File

  Use OpenVMS syntax (defaults to SSL$KEY:SERVER.KEY).

- Certificate File

  Use OpenVMS syntax (defaults to SSL$CRT:SERVER.CRT).

- Country Name
- State or Province Name
- City Name
- Organization Name
- Organization Unit Name
- Common Name

  Common name usage is different for client certificates than it is for server certificates. Generally, the common name on a client certificate is the proper name of the individual requesting a certificate. In the case of server certificates, the common name must be the same as your server's DNS host name (or virtual host name, if name-based virtual hosting is used). Browsers compare the common name in the server certificate with the host name of the server they are connecting to. These must match.

- Email Address
- Display the Certificate

2. View the details of the self-signed certificate (if you chose to display the certificate).

   - Version (SSL 3.0 protocol)
   - Serial number (Certificates issued by a CA have a serial number that is unique to the certificates issued by that CA.)
   - Signature algorithm
   - Issuer
   - Validity (inception and expiration dates)
   - Public key information

Create a Certificate Authority

Creating a certificate authority (CA) allows you to issue certificates using your own private key. The corresponding CA public key is itself contained within a certificate, called a CA Certificate. You must distribute this certificate to clients in order for them to access your server. A browser must contain this CA Certificate in its "trusted root library" in order to trust certificates signed by the CA's private key.

To create a certificate authority, perform the following steps:

1. Enter the information required to create a certificate authority. You must complete all fields to create a valid CA certificate. The certificate request is generated after you respond to the last question.
   - PEM Passphrase
   - Encryption Bits

     The largest recommended size is 1024 bits. Encryption strength is often described in terms of the size of the keys used to perform the encryption; in general, longer keys provide stronger encryption. Key length is measured in bits. Private key sizes larger than 1024 bits are incompatible with some versions of Netscape Navigator and Microsoft Internet Explorer.
   - Default Days

     The default number of days until expiration for certificates issued by the CA. A large number, such as 1825 (5 years) is usually appropriate so that certificates signed with this key do not expire too soon.
   - Certificate Key File

     Use OpenVMS syntax (defaults to SSL$KEY:SERVER_CA.KEY).
   - CA Certificate File

     Use OpenVMS syntax (defaults to SSL$CRT:SERVER_CA.CRT).
   - Country Name

     A certificate authority can define a policy that specifies which distinguished names are optional and which are required. The distinguished name is defined in the config file (.cnf), and is usually made up of more than one field. The number and makeup of the fields are defined by the certificate authority, and are found in the config file under the [req_distinguished_name] field. A certificate authority can also place requirements on the field contents, as can users of certificates. As an example, a Netscape browser requires that the common name for a certificate representing a server has a name that matches a wildcard pattern for the domain name of that server, such as *.xyz.com.
   - State or Province Name
   - City Name
   - Organization Name
   - Organization Unit Name
   - Common Name

     This can be any text string that you want to use to identify the authority. The name can be generic, such as CA Authority, or more specific, such as *nodename*CA.
   - Email Address
   - Display the Certificate
2. View the details of the certificate authority (if you chose to display the certificate).
   - Version (SSL 3.0 protocol)
   - Serial number (Certificates issued by a CA have a serial number that is unique to the certificates issued by that CA.)
   - Signature algorithm
   - Issuer (your distinguished name)
   - Validity (inception and expiration dates)
   - Public key information

Create a Certificate Chain

The following sections describe the steps you must perform to create a certificate chain. Before you create the chain, you must have the following certificates:

- A root CA certificate (See Create a Certificate Authority.)
- One (or more) intermediate CA certificates (See Creating an Intermediate CA (RA) Certificate.)
- Client/server certificate signed with the intermediate CA certificate (See Creating a Client/Server Certificate Signed with an Intermediate CA Certificate.)

Creating an Intermediate CA (RA) Certificate

With the Certificate Tool, you can generate an X509 certificate for an intermediate CA or RA (Registration Authority). Perform the following steps to generate an X509 certificate.

1. Create a certificate signing request. (Select item 3 in the Certificate Tool Main Menu.)
2. Sign the certificate signing request with the root CA certificate. (Select item 6 in the Certificate Tool Main Menu.)

NOTE To create an intermediate CA, you must encrypt the private key when you create the certificate signing request (with PEM passphrase).

Creating a Client/Server Certificate Signed with an Intermediate CA Certificate

After you create an intermediate CA certificate (described in the previous section), create a client/server certificate as follows:

1. Create a certificate signing request. (Select menu item 3 in the Certificate Tool Main Menu.)
2. Sign the certificate signing request with the intermediate CA certificate. (Select menu item 6 in the Certificate Tool Main Menu.)

Encrypting the private key is not required for creating a client/server certificate. However, if the key is encrypted, you can also use the certificate as an intermedicate CA certificate with which another certificate will be signed.

Creating a Certificate Chain File

Some OpenSSL APIs require a certificate chain file. This file contains certificates that form the certificate chain (from the client/server certificate to the root CA certificate).

To create a certificate chain file, append the certificates of intermediate CA(s) and the root CA to the client/server certificate. The order in the file can be expressed as follows:

```
client/server cert >>> intermediate CA1 >>> intermediate CA2 >>> root CA
```

Enter the following command to create a certificate chain file:

```
$ COPY CLIENT_CERT.PEM, INTER_CA1.PEM, INTER_CA2.PEM, -
_$ ROOT_CA.PEM, CERT_CHAIN.PEM
```

Sign a Certificate Signing Request

Signing someone else's certificate signing request is the function of a certificate authority. When you send a signed certificate back, it can be used to start the server with the passphrase they have. Embedded in the certificate is your public key. It must match the public key you distribute to clients using your server.

To sign a certificate signing request, perform the following steps. The certificate is signed after you respond to the last question.

1. Enter the required information to sign a certificate.

NOTE The inception time of a certificate is based on UTC (Coordinated Universal Time). Verify with your system administrator that your computer's UTC is set correctly.

- CA Certificate File specification

  Use OpenVMS syntax (defaults to SSL$CRT:SERVER_CA.CRT).
- CA Certificate Key File specification

  Use OpenVMS syntax (defaults to SSL$KEY:SERVER_CA.KEY).
- Certificate Request File

  Use OpenVMS syntax (defaults to SSL$CRT:SERVER.CSR).
- Signed Request File specification

  Use OpenVMS syntax (defaults to SSL$CRT:SIGNED.CRT).
- Default Days

  The default number of days until the signed certificate expires.
- PEM Passphrase

  This is a verification field only. You must use the same passphrase you used to create the certificate authority (option 5).

2. View the details of the signed certificate (if you chose to display the certificate):

- Version (SSL 3.0 protocol)
- Serial number (Certificates issued by a CA have a serial number that is unique to the certificates issued by that CA.)
- Signature algorithm
- Issuer (your distinguished name)
- Validity (inception and expiration dates)
- Public key information

Revoke a Certificate

You should revoke a certificate if the certificate has been compromised. The security of a certificate can be compromised if, for example, someone has a copy of the private key, or knows the password to your encrypted key.

A certificate can be revoked by the Certificate Authority that issued the certificate. You can also use the HP SSL Certificate Tool to revoke a certificate, if the certificate was created using the Certificate Tool.

To revoke a certificate using the Certificate Tool, perform the following steps:

1. From the Main Menu, select Option 7 - Revoke a Certificate.
2. Enter the filenames of the Certificate Authority (CA) certificate and key.
3. Enter the filename of the certificate to be revoked.
4. Enter the PEM passphrase of the CA's key.

The Certificate Tool marks that certificate as being revoked in its database.

After you revoke the certificate, you must create a certificate revocation list (CRL).

Create a Certificate Revocation List

After you have revoked all known compromised certificates, you should create a Certificate Revocation List (CRL). You can create a CRL using the HP SSL Certificate Tool.

To create a CRL, perform the following steps:

1. From the Main Menu, select Option 8 - Create a Certificate Revocation List.
2. Enter the filenames of the Certificate Authority (CA) certificate and key.
3. Enter the filename of the Certificate Revocation List. This is the file into which the CRL will be written.
4. Enter the number of days until the next CRL will be issued. Certificate Authorities typically issue CRLs on a periodic basis to maintain the current status of the certificates that it has signed.
5. Enter the PEM passphrase of the CA's key.

The Certificate Tool then creates the CRL in the specified file.

Hash Certificates

This command is required to PEM-encode third-party certificate files and files you create using option 5 (which, by default, are named SERVER_CA.CRT).

For example, the mod_ssl directives related to CA certificate management (`SSLCACertificatePath` and `SSLCACertificateFile`) require hashed files.

To hash a certificate or certificate authority, perform the following steps:

1. Enter the name of the path in which you have installed your CA files. For example, if you installed CA files for HP Secure Web Server, the location is APACHE$SPECIFIC:[CONF.SSL_CRT]*.CRT.
2. Press Return to hash the certificate files at the specified location, or at the default location if you did not enter a path.

   You can verify the existence of the hashed file in the directory you selected by entering the following command:

   ```
   $ DIR APACHE$COMMON:[CONF.SSL_CRT]

   Directory APACHE$COMMON:[CONF.SSL_CRT]

   AE0FEDEE.0;4 DELETE_HASH_FILES.COM;1 SERVER_CA.CRT;4

   Total of 3 files.
   ```

Hash Certificate Revocations

This command is required to PEM-encode third-party certificate revocation lists (CRLs) and ones you create using the OpenSSL command line interface. The `mod_ssl` directives related to managing client revocation lists (`SSLCARevocationPath` and `SSLCARevocationFile`) require hashed CRL files.

To hash certificate revocations, perform the following steps:

1. Install a trusted root CA's CRL file, or create your own using the OPENSSL CA command (using the OpenSSL command line interface).
2. Enter the name of the path in which you have installed your CRL files. For example, if you installed CRL files for HP Secure Web Server, the location is APACHE$ROOT:[CONF.SSL_CRL]*.CRL.
3. Press Return to hash the CRL files at the specified location.

   You can verify the existence of the hashed file in the directory you selected by entering the following command:

   ```
   $ DIR APACHE$SPECIFIC:[CONF.SSL_CRL]

   Directory APACHE$SPECIFIC:[CONF.SSL_CRL]

   AE0FEDEE.R0 CA-BUNDLE.CRL DELETE_HASH_FILES.COM

   Total of 3 files.
   ```

4 SSL Programming Concepts

This chapter discusses how to write application programs using SSL on OpenVMS. The SSL library provides APIs supporting three SSL protocols: SSL Version 2 (SSLv2), SSL Version 3 (SSLv3), and TLS Version 1 (TLSv1). You can write an SSL application program in C or C++.

This chapter provides the following information:

- A description of the seven SSL data structures
- How to configure and obtain certificates
- An SSL programming tutorial that shows the implementation of a simple SSL client and server program using SSL APIs

SSL Data Structures

Before you start SSL application development, you should understand the data structures used for SSL APIs, and the relationships between the data structures.

SSL APIs use data structures to hold various types of information about SSL sessions and connections. The most important structures are `SSL_CTX` and `SSL`. Usually, one `SSL_CTX` structure exists per SSL application program, and an `SSL` structure is created every time a new SSL connection is created. An `SSL` structure inherits configuration information from the `SSL_CTX` structure when it is created.

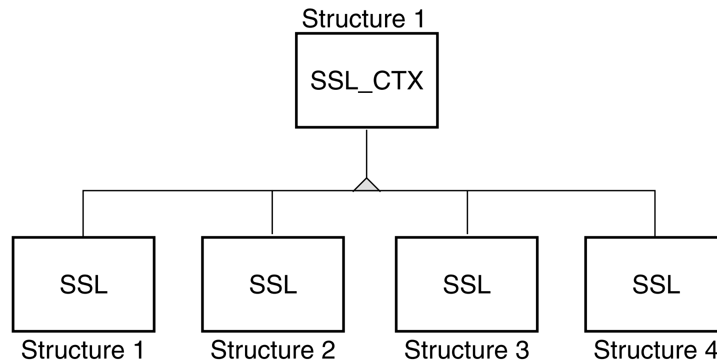
Table 4-1 shows the APIs commonly used for creating and deallocating data structures.

Table 4-1 APIs for Data Structure Creation and Deallocation

Data Structure	API for Creation	API for Deallocation
SSL_CTX	`SSL_CTX_new()`	`SSL_CTX_free()`
SSL	`SSL_new()`	`SSL_free()`
SSL_SESSION	`SSL_SESSION_new()`	`SSL_SESSION_free()`
BIO	`BIO_new()`	`BIO_free()`
X509	`X509_new()`	`X509_free()`
RSA	`RSA_new()`	`RSA_free()`
DH	`DH_new()`	`DH_free()`

Figure 4-1 shows the relationship between the `SSL_CTX` and `SSL` data structures.

Figure 4-1 Relationship Between SSL_CTX and SSL



SSL_CTX Structure

The SSL_CTX structure is defined in ssl.h. An SSL_CTX structure stores default values for SSL structures. (The SSL structures are created after the SSL_CTX structure is created and configured.) The SSL_CTX structure also holds information about SSL connections and sessions (the numbers of new SSL connections, renegotiations, session resumptions, and so on).

Each SSL client or server program creates and keeps only one SSL_CTX structure. The SSL_CTX structure is created at the beginning of the SSL application program. The SSL_CTX structure is configured with the default values that will be inherited by the SSL structures. For example, a CA certificate loaded in the SSL_CTX structure is also loaded into an SSL structure when that SSL structure is created.

NOTE Data structure definitions are subject to change in future releases of HP SSL for OpenVMS.

SSL Structure

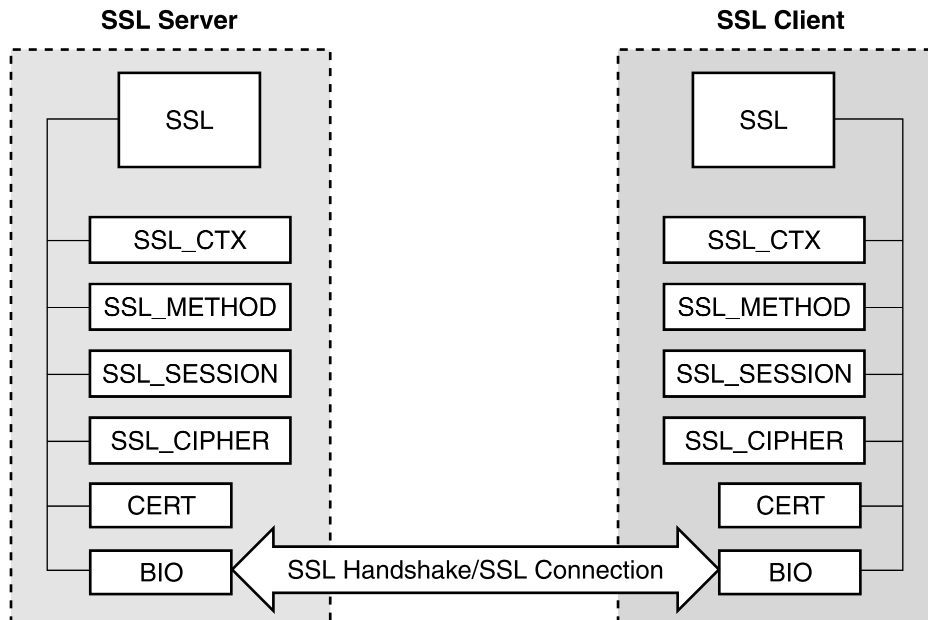
An SSL structure is created for every SSL connection in the SSL client or server program. You create the SSL structure after creating and configuring the SSL_CTX structure because the SSL structure inherits default values from the SSL_CTX structure. The inheritance of the default values enables the SSL structure to be used without explicit configuration. However, it is possible to change the inherited values in a specific SSL structure.

An SSL structure saves the addresses of data structures that store information about SSL connections and sessions. These data structures are as follows:

- The SSL_CTX structure from which the SSL structure is created
- SSL_METHOD (SSL protocol version)
- SSL_SESSION
- SSL_CIPHER
- CERT (certificate information extracted from an X.509 structure)
- BIO (an SSL connection is performed via BIO)

The SSL information (protocol version, connection status values, and so on) in the SSL structure is used for the SSL connection. Figure 4-2 shows the structures associated with the SSL structure.

Figure 4-2 Structures Associated with SSL Structure



SSL_METHOD Structure

The SSL_METHOD structure is defined in **ssl.h**. An SSL_METHOD structure contains pointers to the functions that implement the SSL protocol version specified. This structure must be created before creation of the SSL_CTX structure.

SSL_CIPHER Structure

The SSL_CIPHER structure is defined in the `ssl.h` header file. An SSL_CIPHER structure holds information about the cipher suite used for SSL connections and sessions.

CERT/X509 Structure

In OpenSSL application programs, an X.509 certificate is stored as an X509 structure. However, after loading an X509 structure into an SSL_CTX or SSL structure, the X.509 certificate information is extracted from the X509 structure and stored in a CERT structure associated with the SSL_CTX or SSL structure. The X509 and CERT structures are defined in `x509.h` and `ssl_locl.h`, respectively.

NOTE The `ssl_locl.h` header file is not used for SSL application programs because it defines only internal functions and structures, such as the CERT structure. In SSL application programs, a certificate is stored in an X509 structure, not in a CERT structure. An SSL application developer does not need to know the definition of the CERT structure and `ssl_locl.h`.

BIO Structure

A `BIO` structure is an I/O abstraction in an SSL application with SSL APIs. The `BIO` structure encapsulates an underlying I/O secured by SSL, and all the communication between the client and server is conducted through this structure. The `BIO` structure is defined in `bio.h`.

Certificates for SSL Applications

To establish an SSL connection successfully, you must load proper certificates into the SSL client and server. In this section, a few common uses of certificates are described. For general information about certificates, see Chapter 3.

Configuring Certificates in the SSL Client and Server

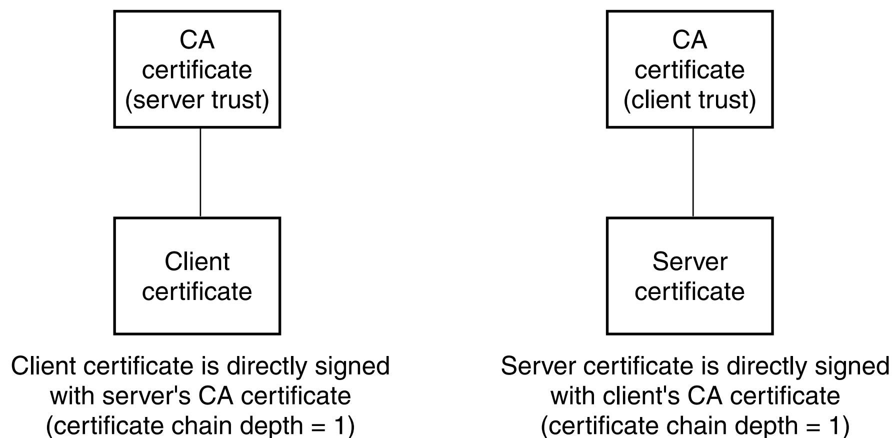
SSL client and server applications might require four certificates:

- Server-s CA certificate
- Client-s CA certificate
- Client certificate
- Server certificate

A **root CA** is a CA certificate that is located as a root in a certificate signing hierarchy. A root CA is not signed by any other CA - it is signed by itself. In Figure 4-3 and Figure 4-4, the CA certificates correspond to root CAs.

For successful certificate verification, the certificates must have the proper signing relationships, as shown in Figure 4-3 and Figure 4-4. In Figure 4-3, the client and server certificates are directly signed by their peers-CAs.

Figure 4-3 Client and Server Certificates Directly Signed by CAs



NOTE The client and server certificates are not necessarily directly signed by the CAs (see Figure 4-3). In some cases, the certificate is signed by an RA (registration authority) or an intermediate CA whose certificate is signed by the CA that is trusted by the peer. (The client certificate in Figure 4-4 is an example of this situation.) In other cases, the certificate's signing chain may involve more RAs or intermediate CAs. (The server certificate in Figure 4-4 is an example of this situation.)

As long as the chain depth setting is appropriate (that is, the certificate chain depth for verification is longer than the depth from the CA to the certificate being verified), the certificate verification succeeds.

Figure 4-4 Client and Server Certificates Indirectly Signed by CAs

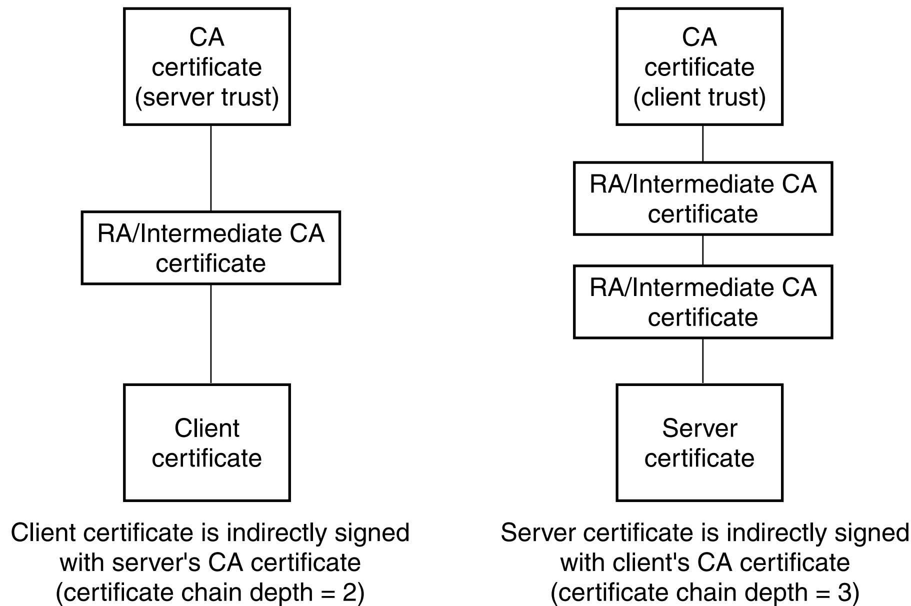
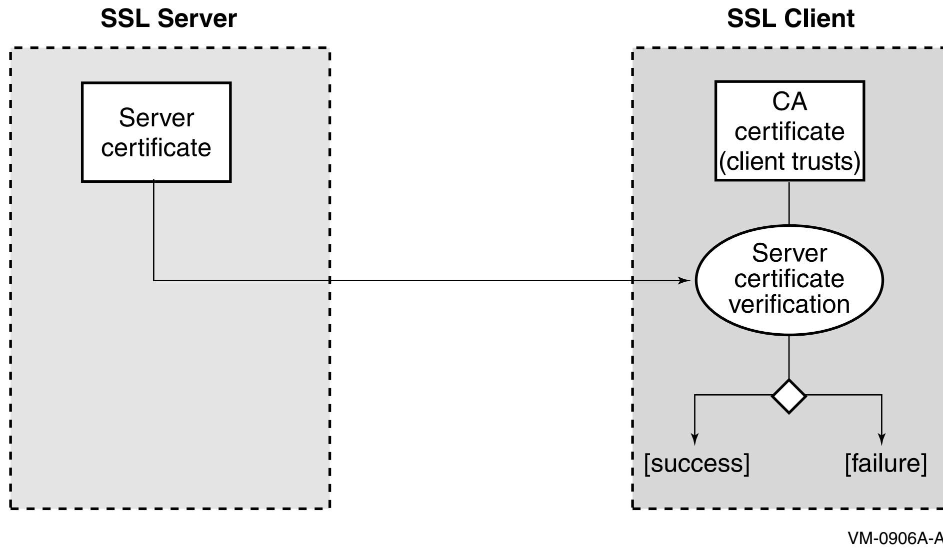


Figure 4-5 depicts the most common deployment of certificates. This deployment is often used when establishing SSL connections between web browsers and a web server. As part of its initialization, the SSL server loads a certificate (server certificate) signed by a CA. This CA is trusted by the SSL clients. When a client verifies the server, the server certificate is sent to the client and then is verified against the CA certificate. The fact that the server has a certificate signed by a trustworthy CA means that the server can be trusted by the client, because the client trusts the CA. This certificate setup prevents the SSL client from establishing an SSL connection with an untrustworthy SSL server.

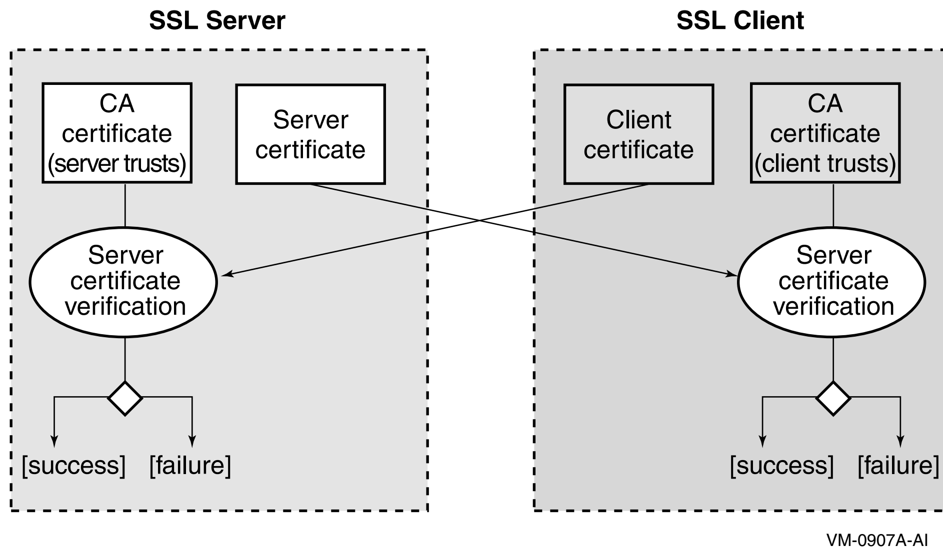
Figure 4-5 Certificates on SSL Client and Server (Case 1)



In addition to server certificate verification on the SSL client, you can perform client certificate verification on the SSL server. This is shown in Figure 4-6. Web sites that require higher security, such as banks and online brokers, deploy this model. The SSL client connecting to this type of SSL server is requested to send its certificate (client certificate) to the server. The SSL server then performs client authentication based on the client certificate verification.

This method is the same as the one used in Figure 4-5, but in this case the server checks the client certificate against the server-s CA certificate to establish the level of trust. Using this implementation, the SSL server can achieve enhanced client authentication by combining with another authentication method, such as requiring a user name and password.

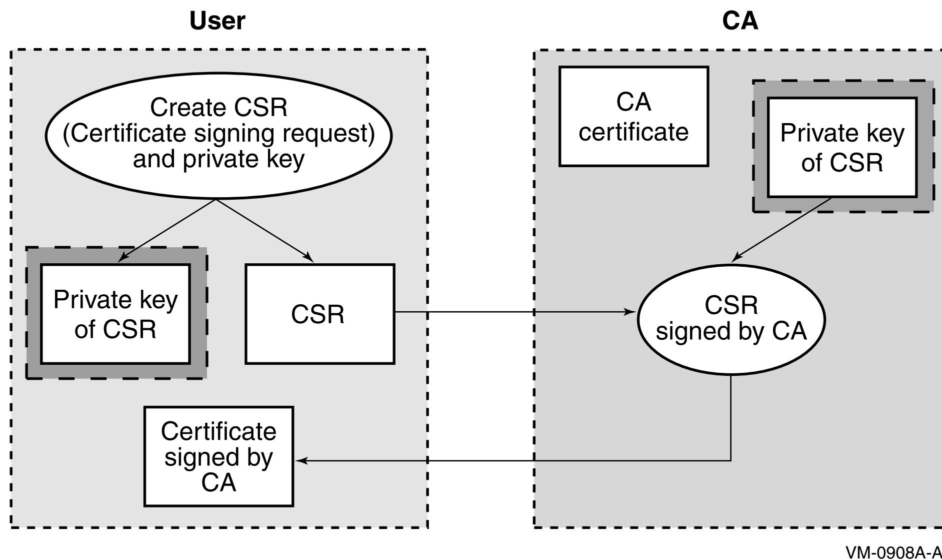
Figure 4-6 Certificates on SSL Client and Server (Case 2)



Obtaining and Creating Certificates

If the proper certificates are not in place, the SSL application user or developer must either create them or obtain them from a trustworthy organization such as a CA or RA. The SSL command line interface (described in Chapter 5) and Certificate Tool (described in Chapter 3) allow you to create X.509 certificates. Figure 4-7 shows the process for creating an X.509 certificate.

Figure 4-7 Certificate Creation Process



When you obtain or create a certificate, consider the following:

- Algorithms
- Key size
- Certificate/key format
- Security policies

Algorithms: RSA certificate with RSA keys or DSA certificate with DH keys

Although RSA certificates are commonly used for SSL, DSA certificates can be loaded in the SSL structure as well. (Most SSL servers load only RSA certificates. SSL servers that use DSA certificates are rare.)

NOTE RSA and DSA certificates and keys are incompatible. An SSL client that has only an RSA certificate and key cannot establish a connection with an SSL server that has only a DSA certificate and key.

To avoid this problem, you can load both RSA and DSA certificates and key pairs in the `SSL_CTX` and SSL structure. (For more information, see the description of the `SSL_CTX_use_certificate()` and `SSL_CTX_set_cipher_list()` APIs in this manual.)

If you use a DSA certificate, you must load DH keys. Although the RSA algorithm is used for both key exchange and signing operations, DSA can be used only for signing. Therefore, DH is used as the key agreement algorithm with a DSA certificate in an SSL application.

NOTE DSA certificates and DH keys cannot be created with the OpenVMS SSL Certificate Tool (described in Chapter 3). Use the SSL command line interface, described in Chapter 5, instead.

Key size: 512-bit, 1024-bit, or others

You must specify the key size of the algorithms when you create a certificate. The key size affects security and performance of the SSL application. A longer key makes the application more secure, but it can slow performance. A shorter key makes encryption and decryption faster, but lowers security.

Usually RSA and DSA keys are 512-bit, 1024-bit or 2048-bit. (1024-bit keys are the most commonly used.) In some cases, you must decide the key size based on the application-s requirement or corporate or national security policy.

Certificate and key formats: PEM, DER or others

The OpenSSL command line interface supports the following three certificate formats:

DER - Encodes the certificate using Distinguished Encoding Rules.
PEM - The Base64 encoding of the DER encoding, with header and footer lines added.
NET - An obsolete Netscape server format.

The most common certificate format for SSL applications is PEM. The SSL Certificate Tool, described in Chapter 3, supports only the PEM format. If a DER certificate is necessary, use the SSL command line interface, described in Chapter 5.

Security policy of the application using the certificates

Check the application-s security policy or requirements when you issue certificates. Some applications require certain attributes or values in the X.509 certificates. For example, SSL applications for financial transactions might have a security policy to use 1024-bit or longer RSA keys, or certain extensions in an X.509 certificates might be mandatory.

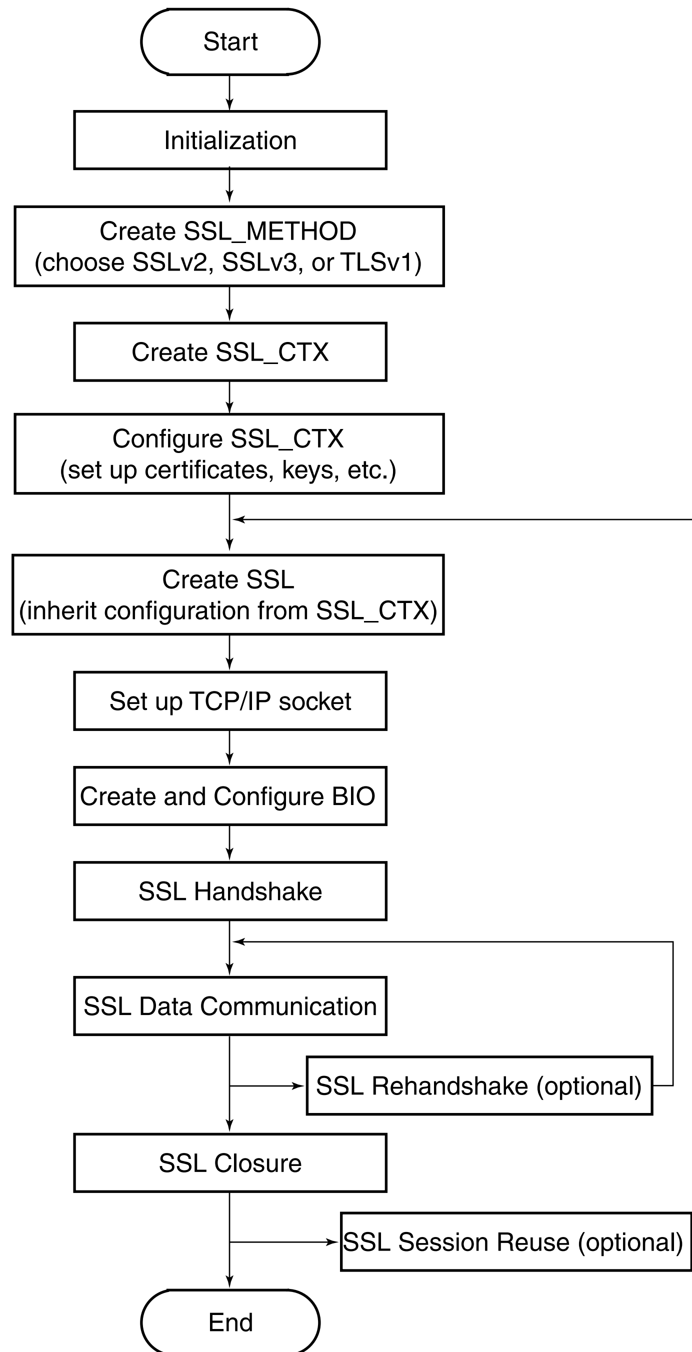
Many countries have national policies regarding encryption. Using and exporting strong encryption algorithms and keys might be affected by these policies. Also, some organizations might have policies that disallow their employees using strong encryption.

SSL Programming Tutorial

This section demonstrates the implementation of a simple SSL client and server program using OpenSSL APIs.

Although SSL client and server programs might differ in their setup and configuration, their common internal procedures can be summarized in Figure 4-8. These procedures are discussed in the following sections.

Figure 4-8 Overview of SSL Application with OpenSSL APIs



Initializing the SSL Library

Before you can call any other OpenSSL APIs in the SSL application programs, you must perform initialization using the following SSL APIs.

```
SSL_library_init(); /* load encryption & hash algorithms for SSL */
SSL_load_error_strings(); /* load the error strings for good error reporting */
```

The `SSL_library_init()` API registers all ciphers and hash algorithms used in SSL APIs. The encryption algorithms loaded with this API are DES-CBC, DES-EDE3-CBC, RC2 and RC4 (IDEA and RC5 are not available in HP SSL for OpenVMS); and the hash algorithms are MD2, MD5, and SHA. The `SSL_library_init()` API has a return value that is always 1 (integer).

SSL applications should call the `SSL_load_error_strings()` API. This API loads error strings for SSL APIs as well as for Crypto APIs. Both SSL and Crypto error strings need to be loaded because many SSL applications call some Crypto APIs as well as SSL APIs.

Creating and Setting Up the SSL Context Structure (SSL_CTX)

The first step after the intialization is to choose an SSL/TLS protocol version. Do this by creating an `SSL_METHOD` structure with one of the following APIs. The `SSL_METHOD` structure is then used to create an `SSL_CTX` structure with the `SSL_CTX_new()` API.

For every SSL/TLS version, there are three types of APIs to create an `SSL_METHOD` structure: one for both client and server, one for server only, and one for client only. SSLv2, SSLv3, and TLSv1 APIs correspond with the same name protocols. Table 4-2 shows the types of APIs.

Table 4-2 Types of APIs for SSL_METHOD Creation

Protocol type	For combined client and server	For a dedicated server	For a dedicated client
SSLv2	`SSLv2_method()`	`SSLv2_server_ method()`	`SSLv2_client_ method()`
SSLv3	`SSLv3_method()`	`SSLv3_server_ method()`	`SSLv3_client_ method()`
TLSv1	`TLSv1_method()`	`TLSv1_server_ method()`	`TLSv1_client_ method()`
SSLv23	`SSLv23_method()`	`SSLv23_server_ method()`	`SSLv23_client_ method()`

NOTE There is no SSL protocol version named SSLv23. The `SSLv23_method()` API and its variants choose SSLv2, SSLv3, or TLSv1 for compatibility with the peer.

Consider the incompatibility among the SSL/TLS versions when you develop SSL client/server applications. For example, a TLSv1 server cannot understand a client-hello message from an SSLv2 or SSLv3 client. The SSLv2 client/server recognizes messages from only an SSLv2 peer. The `SSLv23_method()` API and its variants may be used when the compatibility with the peer is important. An SSL server with the SSLv23 method can understand any of the SSLv2, SSLv3, and TLSv1 hello messages. However, the SSL client using the SSLv23 method cannot establish connection with the SSL server with the SSLv3/TLSv1 method because SSLv2 hello message is sent by the client.

The `SSL_CTX_new()` API takes the `SSL_METHOD` structure as an argument and creates an `SSL_CTX` structure.

In the following example, an `SSL_METHOD` structure that can be used for either an SSLv3 client or SSLv3 server is created and passed to `SSL_CTX_new()`. The `SSL_CTX` structure is initialized for SSLv3 client and server.

```
meth = SSLv3_method();
ctx = SSL_CTX_new(meth);
```

Setting Up the Certificate and Key

Certificates for SSL Applications discussed how the SSL client and server programs require you to set up appropriate certificates. This setup is done by loading the certificates and keys into the `SSL_CTX` or SSL structures. The mandatory and optional certificates are as follows:

- For the SSL server:

  Server's own certificate (mandatory)
  CA certificate (optional)

- For the SSL client:

  CA certificate (mandatory)
  Client's own certificate (optional)

Loading a Certificate (Client/Server Certificate)

Use the `SSL_CTX_use_certificate_file()` API to load a certificate into an `SSL_CTX` structure. Use the `SSL_use_certificate_file()` API to load a certificate into an `SSL` structure. When the `SSL` structure is created, the `SSL` structure automatically loads the same certificate that is contained in the `SSL_CTX` structure. Therefore, you onlyneed to call the `SSL_use_certificate_file()` API for the `SSL` structure only if it needs to load a different certificate than the default certificate contained in the `SSL_CTX` structure.

Loading a Private Key

The next step is to set a private key that corresponds to the server or client certificate. In the SSL handshake, a certificate (which contains the public key) is transmitted to allow the peer to use it for encryption. The encrypted message sent from the peer can be decrypted only using the private key. You must preload the private key that was created with the public key into the `SSL` structure.

The following APIs load a private key into an `SSL` or `SSL_CTX` structure:

- `SSL_CTX_use_PrivateKey()`
- `SSL_CTX_use_PrivateKey_ASN1()`
- `SSL_CTX_use_PrivateKey_file()`
- `SSL_CTX_use_RSAPrivateKey()`
- `SSL_CTX_use_RSAPrivateKey_ASN1()`
- `SSL_CTX_use_RSAPrivateKey_file()`
- `SSL_use_PrivateKey()`
- `SSL_use_PrivateKey_ASN1()`
- `SSL_use_PrivateKey_file()`
- `SSL_use_RSAPrivateKey()`
- `SSL_use_RSAPrivateKey_ASN1()`
- `SSL_use_RSAPrivateKey_file()`

Loading a CA Certificate

To verify a certificate, you must first load a CA certificate (because the peer certificate is verified against a CA certificate). The `SSL_CTX_load_verify_locations()` API loads a CA certificate into the `SSL_CTX` structure.

The prototype of this API is as follows:

```
int SSL_CTX_load_verify_locations(SSL_CTX *ctx, const char *CAfile,
const char *CApath);
```

The first argument, `ctx`, points to an `SSL_CTX` structure into which the CA certificate is loaded. The second and third arguments, `CAfile` and `CApath`, are used to specify the location of the CA certificate. When looking up CA certificates, the OpenSSL library first searches the certificates in `CAfile`, then those in `CApath`.

The following rules apply to the `CAfile` and `CApath` arguments:

- If the certificate is specified by `CAfile` (the certificate must exist in the same directory as the SSL application), specify NULL for `CApath`.
- To use the third argument, `CApath`, specify NULL for `CAfile`. You must also hash the CA certificates in the directory specified by `CApath`. Use the Certificate Tool (described in Chapter 3) to perform the hashing operation.

Setting Up Peer Certificate Verification

The CA certificate loaded in the `SSL_CTX` structure is used for peer certificate verification. For example, peer certificate verification on the SSL client is performed by checking the relationships between the CA certificate (loaded in the SSL client) and the server certificate.

For successful verification, the peer certificate must be signed with the CA certificate directly or indirectly (a proper certificate chain exists). The certificate chain length from the CA certificate to the peer certificate can be set in the `verify_depth` field of the `SSL_CTX`and `SSL` structures. (The value in `SSL` is inherited from `SSL_CTX` when you create an `SSL` structure using the `SSL_new()` API). Setting `verify_depth` to 1 means that the peer certificate must be directly signed by the CA certificate.

The `SSL_CTX_set_verify()` API allows you to set the verification flags in the `SSL_CTX` structure and a callback function for customized verification as its third argument. (Setting NULL to the callback function means the built-in default verification function is used.) In the second argument of `SSL_CTX_set_verify()`, you can set the following macros:

- `SSL_VERIFY_NONE`
- `SSL_VERIFY_PEER`
- `SSL_VERIFY_FAIL_IF_NO_PEER_CERT`
- `SSL_VERIFY_CLIENT_ONCE`

The `SSL_VERIFY_PEER` macro can be used on both SSL client and server to enable the verification. However, the subsequent behaviors depend on whether the macro is set on a client or a server. For example:

```
/* Set a callback function (verify_callback) for peer certificate */
/* verification */
SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, verify_callback);
/* Set the verification depth to 1 */
SSL_CTX_set_verify_depth(ctx,1);
```

You can verify a peer certificate in another, less common way - by using the `SSL_get_verify_result()` API. This method allows you to obtain the peer certificate verification result without using the `SSL_CTX_set_verify()` API.

Call the following two APIs *before* you call the `SSL_get_verify_result()` API:

1. Call `SSL_connect()` (in the client) or `SSL_accept()` (in the server) to perform the SSL handshake. Certificate verification is performed during the handshake. `SSL_get_verify_result()` cannot obtain the result before the verification process.
2. Call `SSL_get_peer_certificate()` to explicitly obtain the peer certificate. The `X509_V_OK` macro value is returned when a peer certificate is not presented as well as when the verification succeeds.

The following code shows how to use `SSL_get_verify_result()` in the SSL client:

```
 SSL_CTX_set_verify_depth(ctx, 1);
 err = SSL_connect(ssl);
 if(SSL_get_peer_certificate(ssl) != NULL)
      {
      if(SSL_get_verify_result(ssl) == X509_V_OK)
 BIO_printf(bio_c_out, "client verification with SSL_get_verify_result()
      succeeded.\n");
      else{

BIO_printf(bio_err, "client verification with SSL_get_verify_result()
      failed.\n");

 exit(1);
      }
      }
 else
      BIO_printf(bio_c_out, -the peer certificate was not presented.\n-);
```

Example 1: Setting Up Certificates for the SSL Server

The SSL protocol requires that the server set its own certificate and key. If you want the server to conduct client authentication with the client certificate, the server must load a CA certificate so that it can verify the client-s certificate.

The following example shows how to set up certificates for the SSL server:

```
 /* Load server certificate into the SSL context */
        if (SSL_CTX_use_certificate_file(ctx, SERVER_CERT,
         SSL_FILETYPE_PEM) <= 0) }
                ERR_print_errors(bio_err);  /* ==
                  ERR_print_errors_fp(stderr); */
                exit(1);
        }

        /* Load the server private-key into the SSL context */
        if (SSL_CTX_use_PrivateKey_file(ctx, SERVER_KEY,
          SSL_FILETYPE_PEM) <= 0) {
                ERR_print_errors(bio_err);  /* ==
                  ERR_print_errors_fp(stderr); */
                exit(1);
        }

/* Load trusted CA. */
        if (!SSL_CTX_load_verify_locations(ctx,CA_CERT,NULL)) {
                ERR_print_errors(bio_err);  /* ==
                  ERR_print_errors_fp(stderr); */
                exit(1);
        }

        /* Set to require peer (client) certificate verification */
        SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, verify_callback);
       /* Set the verification depth to 1 */
        SSL_CTX_set_verify_depth(ctx,1);
```

Example 2: Setting Up Certificates for the SSL Client

Generally, the SSL client verifies the server certificate in the process of the SSL handshake. This verification requires the SSL client to set up its trusting CA certificate. The server certificate must be signed with the CA certificate loaded in the SSL client in order for the server certificate verification to succeed.

The following example shows how to set up certificates for the SSL client:

```
/*----- Load a client certificate into the SSL_CTX structure -----*/
       if(SSL_CTX_use_certificate_file(ctx,CLIENT_CERT,
       SSL_FILETYPE_PEM) <= 0){
              ERR_print_errors_fp(stderr);
              exit(1);
       }

/*----- Load a private-key into the SSL_CTX structure -----*/
        if(SSL_CTX_use_PrivateKey_file(ctx,CLIENT_KEY,
        SSL_FILETYPE_PEM) <= 0){
              ERR_print_errors_fp(stderr);
              exit(1);
        }

/* Load trusted CA. */
        if (!SSL_CTX_load_verify_locations(ctx,CA_CERT,NULL)) {
                ERR_print_errors_fp(stderr);
                exit(1);
        }
```

Creating and Setting Up the SSL Structure

Call `SSL_new()` to create an `SSL` structure. Information for an SSL connection is stored in the `SSL` structure. The protocol for the `SSL_new()` API is as follows:

```
ssl = SSL_new(ctx);
```

A newly created `SSL` structure inherits information from the `SSL_CTX` structure. This information includes types of connection methods, options, verification settings, and timeout settings. No additional settings are required for the `SSL` structure if the appropriate initialization and configuration have been done for the `SSL_CTX` structure.

You can modify the default values in the `SSL` structure using SSL APIs. To do this, use variants of the APIs that set attributes of the `SSL_CTX` structure. For example, you can use `SSL_CTX_use_certificate()` to load a certificate into an `SSL_CTX` structure, and you can use `SSL_use_certificate()` to load a certificate into an `SSL` structure.

Setting Up the TCP/IP Connection

Although SSL works with some other reliable protocols, TCP/IP is the most common transport protocol used with SSL.

The following sections describe how to set up TCP/IP for the SSL APIs. This configuration is the same as in many other TCP/IP client/server application programs; it is not specific to SSL API applications. In these sections, TCP/IP is set up with the ordinary socket APIs, although it is also possible to use OpenVMS system services.

Creating and Setting Up the Listening Socket (on the SSL Server)

The SSL server needs two sockets as an ordinary TCP/IP server-one for the SSL connection, the other for detecting an incoming connection request from the SSL client.

In the following code, the `socket()` function creates a listening socket. After the address and port are assigned to the listening socket with `bind()`, the `listen()` function allows the listening socket to handle an incoming TCP/IP connection request from the client.

```
listen_sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
CHK_ERR(listen_sock, "socket");

memset(&sa_serv, 0, sizeof(sa_serv));
sa_serv.sin_family      = AF_INET;
sa_serv.sin_addr.s_addr = INADDR_ANY;
sa_serv.sin_port        = htons(s_port);      /* Server Port number */

err = bind(listen_sock, (struct sockaddr*)&sa_serv,sizeof(sa_serv));
CHK_ERR(err, "bind");

/* Receive a TCP connection. */
err = listen(listen_sock, 5);
CHK_ERR(err, "listen");
```

Creating and Setting Up the Socket (on the SSL Client)

On the client, you must create a TCP/IP socket and attempt to connect to the server with this socket. To establish a connection to the specified server, the TCP/IP `connect()` function is used. If the function succeeds, the socket passed to the `connect()` function as a first argument can be used for data communication over the connection.

```
sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
CHK_ERR(sock, "socket");

memset (&server_addr, '\0', sizeof(server_addr));
server_addr.sin_family      = AF_INET;
server_addr.sin_port        = htons(s_port);       /* Server Port number */
server_addr.sin_addr.s_addr = inet_addr(s_ipaddr); /* Server IP */

err = connect(sock, (struct sockaddr*) &server_addr, sizeof(server_addr));
CHK_ERR(err, "connect");
```

Establishing a TCP/IP Connection (on the SSL Server)

To accept an incoming connection request and to establish a TCP/IP connection, the SSL server needs to call the `accept()` function. The socket created with this function is used for the data communication between the SSL client and server. For example:

```
sock = accept(listen_sock, (struct sockaddr*)&sa_cli, &client_len);
BIO_printf(bio_c_out, "Connection from %lx, port %x\n",
sa_cli.sin_addr.s_addr, sa_cli.sin_port);
```

Setting Up the Socket/Socket BIO in the SSL Structure

After you create the `SSL` structure and the TCP/IP socket (`sock`), you must configure them so that SSL data communication with the `SSL` structure can be performed automatically through the socket.

The following code fragments show the various ways to assign `sock` to `ssl`. The simplest way is to set the socket directly into the SSL structure, as follows:

```
SSL_set_fd(ssl, sock);
```

A better way is to use a BIO structure, which is the I/O abstraction provided by OpenSSL. This way is preferable because BIO hides details of an underlying I/O. As long as a BIO structure is set up properly, you can establish SSL connections over any I/O.

The following two examples demonstrate how to create a socket BIO and set it into the SSL structure.

```
sbio=BIO_new(BIO_s_socket());
BIO_set_fd(sbio, sock, BIO_NOCLOSE);
SSL_set_bio(ssl, sbio, sbio);
```

In the following example, the BIO_new_socket() API creates a socket BIO in which the TCP/IP socket is assigned, and the SSL_set_bio() API assigns the socket BIO into the SSL structure. The following two lines of code are equivalent to the preceding three lines:

```
sbio = BIO_new_socket(socket, BIO_NOCLOSE);
SSL_set_bio(ssl, sbio, sbio);
```

NOTE If there is already a BIO connected to ssl, BIO_free() is called (for both the reading and writing side, if different).

SSL Handshake

The SSL handshake is a complicated process that involves significant cryptographic key exchanges. However, the handshake can be completed by calling SSL_accept() on the SSL server and SSL_connect() on the SSL client.

SSL Handshake on the SSL Server

The SSL_accept() API waits for an SSL handshake initiation from the SSL client. Successful completion of this API means that the SSL handshake has been completed.

```
err = SSL_accept(ssl);
```

SSL Handshake on the SSL Client

The SSL client calls the SSL_connect() API to initiate an SSL handshake. If this API returns a value of 1, the handshake has completed successfully. The data can now be transmitted securely over this connection.

```
err = SSL_connect(ssl);
```

Performing an SSL Handshake with SSL_read and SSL_write (Optional)

Optionally, you can call SSL_write() and SSL_read() to complete the SSL handshake as well as perform SSL data exchange. With this approach, you must call SSL_set_accept_state() before you call SSL_read() on the SSL server. You must also call SSL_set_connect_state()before you call SSL_write() on the client. For example:

```
/* When SSL_accept() is not called, SSL_set_accept_state() */
/* must be called prior to SSL_read() */
SSL_set_accept_state(ssl);

/* When SSL_connect() is not called, SSL_set_connect_state() */
/* must be called prior to

SSL_write() */
SSL_set_connect_state(ssl);
```

Obtaining a Peer Certificate (Optional)

Optionally, after the SSL handshake, you can obtain a peer certificate by calling `SSL_get_peer_certificate()`. This API is often used for straight certificate verification, such as checking certificate information (for example, the common name and expiration date).

```
peer_cert = SSL_get_peer_certificate(ssl);
```

Transmitting SSL Data

After the SSL handshake is completed, data can be transmitted securely over the established SSL connection. `SSL_write()` and `SSL_read()` are used for SSL data transmission, just as `write()` and `read()` or `send()` and `recv()` are used for an ordinary TCP/IP connection.

Sending Data

To send data over the SSL connection, call `SSL_write()`. The data to be sent is stored in the buffer specified as a second argument. For example:

```
err = SSL_write(ssl, wbuf, strlen(wbuf));
```

Receiving Data

To read data sent from the peer over the SSL connection, call `SSL_read()`. The received data is stored in the buffer specified as a second argument. For example:

```
err = SSL_read(ssl, rbuf, sizeof(rbuf)-1);
```

Using BIOs for SSL Data Transmission (Optional)

Instead of using `SSL_write()` and `SSL_read()`, you can transmit data by calling `BIO_puts()` and `BIO_gets()`, and `BIO_write()` and `BIO_read()`, provided that a buffer BIO is created and set up as follows:

```
BIO *buf_io, *ssl_bio;
charrbuf[READBUF_SIZE];
charwbuf[WRITEBUF_SIZE]

buf_io = BIO_new(BIO_f_buffer());/* create a buffer BIO */
ssl_bio = BIO_new(BIO_f_ssl());  /* create an ssl BIO */
BIO_set_ssl(ssl_bio, ssl, BIO_CLOSE);/* assign the ssl BIO to SSL */
BIO_push(buf_io, ssl_bio);/* add ssl_bio to buf_io */

ret = BIO_puts(buf_io, wbuf);
/* Write contents of wbuf[] into buf_io */
ret = BIO_write(buf_io, wbuf, wlen);
/* Write wlen-byte contents of wbuf[] into buf_io */

ret = BIO_gets(buf_io, rbuf, READBUF_SIZE);
/* Read data from buf_io and store in rbuf[] */
ret = BIO_read(buf_io, rbuf, rlen);
/* Read rlen-byte data from buf_io and store rbuf[] */
```

Closing an SSL Connection

When you close an SSL connection, the SSL client and server send `close_notify` messages to notify each other of the SSL closure. You use the `SSL_shutdown()` API to send the `close_notify` alert to the peer.

The shutdown procedure consists of two steps:

- Sending a `close_notify` shutdown alert
- Receiving a `close_notify` shutdown alert from the peer

The following rules apply to closing an SSL connection:

- Either party can initiate a close by sending a `close_notify` alert.
- Any data received after sending a closure alert is ignored.
- Each party is required to send a `close_notify` alert before closing the write side of the connection.
- The other party is required both to respond with a `close_notify` alert of its own and to close down the connection immediately, discarding any pending writes.
- The initiator of the close is not required to wait for the responding `close_notify` alert before closing the read side of the connection.

The SSL client or server that initiates the SSL closure calls `SSL_shutdown()` either once or twice. If it calls the API twice, one call sends the `close_notify` alert and one call receives the response from the peer. If the initator calls the API only once, the initiator does not receive the `close_notify` alert from the peer. (The initiator is not required to wait for the responding alert.)

The peer that receives the alert calls `SSL_shutdown()` once to send the alert to the initiating party.

Resuming an SSL Connection

You can reuse the information from an already established SSL session to create a new SSL connection. Because the new SSL connection is reusing the same master secret, the SSL handshake can be performed more quickly. As a result, SSL session resumption can reduce the load of a server that is accepting many SSL connections.

Perform the following steps to resume an SSL session on the SSL client:

1. Start the first SSL connection. This also creates an SSL session.

```
ret = SSL_connect(ssl)
(Use SSL_read() / SSL_write() for data communication
      over the SSL connection)
```

2. Save the SSL session information.

```
sess = SSL_get1_session(ssl);
/* sess is an SSL_SESSION, and ssl is an SSL */
```

3. Shut down the first SSL connection.

```
SSL_shutdown(ssl);
```

4. Create a new SSL structure.

```
ssl = SSL_new(ctx);
```

5. Set the SSL session to a new SSL session before calling `SSL_connect()`.

```
SSL_set_session(ssl, sess);
err = SSL_connect(ssl);
```

6. Start the second SSL connection with resumption of the session.

```
ret = SSL_connect(ssl)
(Use SSL_read() / SSL_write() for data communication
over the SSL connection)
```

If the SSL client calls `SSL_get1_session()` and `SSL_set_session()`, the SSL server can accept a new SSL connection using the same session without calling special APIs to resume the session. The server does this by following the steps discussed in Creating and Setting Up the SSL Structure, Setting Up the TCP/IP Connection, Setting Up the Socket/Socket BIO in the SSL Structure, SSL Handshake, and Transmitting SSL Data.

NOTE Calling `SSL_free()` results in the failure of the SSL session to resume, even if you saved the SSL session with `SSL_get1_session()`.

Renegotiating the SSL Handshake

SSL renegotiation is a new SSL handshake over an already established SSL connection. Because the renegotiation messages (including types of ciphers and encryption keys) are encrypted and then sent over the existing SSL connection, SSL renegotiation can establish another SSL session securely. SSL renegotiation is useful in the following situations, once you have established an ordinary SSL session:

- When you require client authentication
- When you are using a different set of encryption and decryption keys
- When you are using a different set of encryption and hashing algorithms

SSL renegotiation can be initiated by either the SSL client or the SSL server. Initiating an SSL renegotiation on the client requires a different set of APIs (on both the initiating SSL client and the accepting server) from the APIs required for the initiation on the SSL server (in this case, on the initiating SSL server and the accepting SSL client).

The following sections discuss the required APIs for both situations.

NOTE SSLv2 cannot perform SSL renegotiation. Use SSLv3 or TLSv3 for this operation.

SSL Renegotiation Initiated by the SSL Server

To initiate an SSL renegotiation from the SSL server, call `SSL_renegotiate()` once and `SSL_do_handshake()` twice.

The `SSL_renegotiate()` API sets flags for SSL renegotiation. This API does not actually initiate the renegotiation. The flags turned on by `SSL_renegotiate()` inform `SSL_do_handshake()` that it needs to perform SSL renegotiation with the SSL client. The `SSL_do_handshake()` API performs an actual SSL handshake. The first call sends a -Server Hello- message to the SSL client.

If the first call succeeds, the client has agreed to perform an SSL renegotiation. The server then sets the `SSL_ST_ACCEPT` state in the SSL structure and calls `SSL_do_handshake()` again to complete the rest of the renegotiation.

The following code fragment shows how these APIs are used:

```
printf("Starting SSL renegotiation on SSL server (initiating by SSL server)");
if(SSL_renegotiate(ssl) <= 0){
printf("SSL_renegotiate() failed\n");
exit(1);
}

if(SSL_do_handshake(ssl) <= 0){
printf("SSL_do_handshake() failed\n");
exit(1);
```

```
}

ssl->state = SSL_ST_ACCEPT;

if(SSL_do_handshake(ssl) <= 0){
printf("SSL_do_handshake() failed\n");
exit(1);
}
```

The following code shows the APIs called by the SSL client when the renegotiation is initiated by the server:

```
printf("Starting SSL renegotiation on SSL client (initiating by SSL server)");
/* SSL   renegotiation */
err = SSL_read(ssl, buf, sizeof(buf)-1);
```

As the example shows, `SSL_READ()` performs data exchange, and can also handle connection-related functions such as renegotiation.

SSL Renegotiation Initiated by the SSL Client

The SSL client can also initiate SSL renegotiation. In this case, the setup on the client initiating the renegotiation is similar to that on a server initiating the renegotiation. To complete this operation, the SSL client calls `SSL_renegotiate()` and `SSL_do_handshake()` only once. `SSL_renegotiate()` simply sets the flags for SSL renegotiation, and a single call of `SSL_do_handshake()` covers the entire renegotiation.

```
printf("Starting SSL renegotiation on SSL client (initiating by SSL client)");
if(SSL_renegotiate(ssl) <= 0){
       printf("SSL_renegotiate() failed\n");
       exit(1);
}
if(SSL_do_handshake(ssl) <= 0){
       printf("SSL_do_handshake() failed\n");
       exit(1);
}
```

The following code shows the APIs called by the SSL server when the renegotiation is initiated by the client. (These are the same APIs that are called by the SSL client when the renegotiation is initiated by the server.)

```
printf("Starting SSL renegotiation on SSL server (initiating by SSL client)");
/* SSL renegotiation */
err = SSL_read(ssl, buf, sizeof(buf)-1);
```

Again in this example, `SSL_READ()` is handling the data exchange and connection renegotiation.

Finishing the SSL Application

When you finish an SSL application program, the major task is to free (deallocate) the data structures that were created and used in the application program. The APIs for deallocation usually contain the `_free` suffix, whereas the APIs that create a new data structure contain the`_new` suffix.

You must free data structures that you explicitly created in the SSL application program. Data structures that were created inside another structure with an `xxx_new()` API are automatically deallocated when the structure is deallocated with the corresponding `xxx_free()` API. For example, a BIO structure created with `SSL_new()` is freed when you call `SSL_free()`; you do not need to call `BIO_free()` to free the BIO inside the SSL structure. However, if the application program called `BIO_new()` to allocate a BIO structure, you must free that structure with `BIO_free()`.

NOTE You must call `SSL_shutdown()` before you call `SSL_free()`.

5 OpenSSL Command Line Interface

HP SSL for OpenVMS provides a command line interface that allows you to use the cryptography functions of SSL's cryptography library from the OpenSSL command prompt (OPENSSL>). You can use the command-line interface for the following tasks:

- Creating RSA, DH and DSA key parameters
- Creating X.509 certificates, CSRs, and CRLs
- Calculating message digests
- Encrypting and decrypting with ciphers
- Testing on SSL/TLS clients and servers
- Handling of S/MIME signed or encrypted mail

Command-Line Help

HP SSL for OpenVMS includes three pseudocommands that function like command-line help. When you enter one of these pseudocommands at the OpenSSL prompt, SSL displays a list (one entry per line) of names of all the standard commands, message digest commands, or cipher commands, that are available in the command line interface.

NOTE To use these commands, you must have previously run SYS$STARTUP:SSL$STARTUP.COM and SSL$COM:SSL$UTILS.COM.

The pseudocommands are as follows:

```
$ openssl
openssl> list-standard-commands
openssl> list-message-digest-commands
openssl> list-cipher-commands
```

To obtain a list of all of the commands available, enter the following:

```
$ openssl ?
```

SSL$UTILS.COM sets up foreign commands to provide command-line accesss to the standard, message digest, and cipher commands. You can also display the UNIX manpage documentation for each command by entering the following:

```
$ openssl command-name ?
```

where *command-name* is the name of an OpenSSL command such as `asn1parse`.

Standard Commands

The following are the OpenSSL standard commands.

`asn1parse`
: Parse an ASN.1 sequence

`ca`
: Certificate Authority (CA) Management

`ciphers`
: Cipher Suite Description Determination

`crl`
: Certificate Revocation List (CRL) Management

`crl2pkcs7`
: CRL to PKCS#7 Conversion

`dgst`
: Message Digest Calculation

`dh`
: Diffie-Hellman Parameter Management Obsoleted by `dHParam`.

`dHParam`
: Generation and Management of Diffie-Hellman Parameters

`dsa`
: DSA Data Management

`dsaparam`
: DSA Parameter Generation

`enc`
: Encoding with Ciphers

`errstr`
: Error Number to Error String Conversion

`gendh`
: Generation of Diffie-Hellman Parameters. Obsoleted by `dHParam`.

`gendsa`
: Generation of DSA Parameters

`genrsa`
: Generation of RSA Parameters

`nseq`
: Netscape Certificate Sequence Utility

`passwd`

Generation of hashed passwords

`pkcs12`

PKCS#12 Data Management

`pkcs7`

PKCS#7 Data Management

`pkcs8`

PKCS#8 Data Management

`rand`

Generate pseudo-random bytes

`req`

X.509 Certificate Signing Request (CSR) Management

`rsa`

RSA Data Management

`rsautl`

RSA utility for signing, verification, encryption, and decryption

`s_client`

Implements a generic SSL/TLS client that can establish a transparent connection to a remote server speaking SSL/TLS. This command, however, is intended for testing purposes only and provides only rudimentary interface functionality. Internally, however, it uses most of the functionality of the OpenSSL `ssl` library.

`s_server`

Implements a generic SSL/TLS server that accepts connections from remote clients speaking SSL/TLS. It is intended for testing purposes only and provides only rudimentary interface functionality. Internally, however, it uses most of the functionality of the OpenSSL `ssl` library. It provides both its own command-line oriented protocol for testing SSL functions and a simple HTTP response facility to emulate an SSL/TLS-aware web server.

`s_time`

SSL Connection Timer

`sess_id`

SSL Session Data Management

`smime`

S/MIME mail processing

`speed`

Algorithm Speed Measurement

`spkac`

Signed public key and challenge

`verify`

X.509 Certificate Verification

`version`

OpenSSL Version Information

`x509`

X.509 Certificate Data Management

Message Digest Commands

The following are the OpenSSL message digest commands.

`md2`

MD2 Digest

`md4`

MD4 Digest

`md5`

MD5 Digest

`mdc2`

MDC2 Digest

`rmd160`

RMD-160 Digest

`sha`

SHA Digest

`sha1`

SHA-1 Digest

Encoding and Cipher Commands

The following are the OpenSSL encoding and cipher commands. These commands use the following abbreviations:

- CBC - Cipher Block Chaining
- CFB - Cipher Feedback
- ECB - Electronic Cookbook
- OFB - Output Feedback
- EDE - Encrypt-Decrypt-Encrypt

`base64`

Base64 Encoding

`bf-cbc`

Blowfish in CBC mode

`bf`

Alias for bf-cbc

`bf-cfb`

Blowfish in CFB mode

`bf-ecb`

Blowfish in ECB mode

`bf-ofb`

Blowfish in OFB mode

`cast-cbc`

CAST Cipher in CBC mode

`cast5-cbc`

CAST5 Cipher in CBC mode

`cast`

Alias for cast-cbc

`cast5-cfb`

CAST5 in CFB mode

`cast5-ecb`

CAST5 in ECB mode

`cast5-ofb`

CAST5 in OFB mode

`des-cbc`

DES Cipher in CBC mode

`des`

Alias for des-cbc

`des-cfb`

DES in CFB mode

`des-ofb`

DES in OFB mode

`des-ecb`

DES in ECB mode

`des-ede-cbc`

Two key triple DES EDE in CBC mode

`des-ede`
: Alias for des-ede

`des-ede-cfb`
: Two key triple DES EDE in CFB mode

`des-ede-ofb`
: Two key triple DES EDE in OFB mode

`des-ede3-cbc`
: Three key triple DES EDE in CBC mode

`des-ede3`
: Alias for des-ede3-cbc

`des3`
: Alias for des-ede3-cbc

`des-ede3-cfb`
: Three key triple DES EDE CFB mode

`des-ede3-ofb`
: Three key triple DES EDE in OFB mode

`desx`
: DESX algorithm

`rc2-cbc`
: 128-bit RC2 Cipher in CBC mode

`rc2`
: Alias for rc2-cbc

`rc2-cfb`
: 128-bit RC2 in CFB mode

`rc2-ecb`
: 128-bit RC2 in ECB mode

`rc2-ofb`
: 128-bit RC2 in OFB mode

`rc2-64-cbc`
: 64-bit RC2 in CBC mode

`rc2-40-cbc`
: 40-bit RC2 in CBC mode

`rc4`
: 128-bit RC4 Cipher

`rc4-40`
: 40-bit RC4

Password Arguments

Several commands accept password arguments, typically using the `passin` and the `passout` options, respectively, for input and output passwords. These arguments allow the password to be obtained from a variety of sources. Both options take a single argument in the following format. If no password argument is given and a password is required, then the user is prompted to enter a password. The password is read from the current terminal with echoing turned off.

`pass:password`

The actual password is `password`. Since the password is visible to utilities (such as the `ps` utility in UNIX), use this form only when security is not important.

`env:var`

Obtains the password from the environment variable `var`. Because the environment of other processes is visible on certain platforms (such as `ps` in certain UNIX operating systems), use this option with caution.

`file:pathname`

The first line of `pathname` is the password. If the same `pathname` argument is supplied to the `passin` and `passout` arguments, then the first line is used for the input password and the next line is used for the output password. The `pathname` need not refer to a regular file; for example, it could refer to a device or named pipe.

`fd:number`

Reads the password from the file descriptor `number`. This can be used, for example, to send the data via a pipe.

`stdin`

Reads the password from standard input.

Creating a DH Parameter (Key) File and a DSA Certificate and Key

In order to establish an SSL connection with the DH (key exchange) and DSA (DSS, signing) algorithms, a DH parameter file and DSA certificates and keys are required in your SSL application. The Certificate Tool (described in Chapter 3) does not provide this functionality. However, the OpenSSL command-line utility allows you to create the required files.

The following lines demonstrate how to create the DH and DSA related files.

```
## Create a DH parameter (key size is 1024 bits)
   $ openssl dHParam -outform PEM -out dHParam.pem 1024

## Create a DSA certificate

- Create DSA parameters (key size is 1024 bits)
  $ openssl dsaparam -out dsaparam.pem 1024

- Create a DSA CA certificate and private key(using DSA parameter in dsaparam.pem)
```

```
 $ openssl req -x509 -newkey dsa:dsaparam.pem
-keyout dsa_ca.key -out dsa_ca.crt -config SSL$CONF

- Create DSA certificate signing request(dsa_cert.csr)& private key(dsa_cert.key)

  $ openssl req -out dsa_cert.csr -keyout dsa_cert.key
-newkey dsa:DSAPARAM.PEM -config SSL$CONF

- Sign Certificate Signing Request with DSA CA Certificate and Create a New Certificate

 $ openssl ca -in dsa_cert.csr -out dsa_cert.crt

-config SSL$CA_CONF
```

6 Sample Programs

The HP SSL for OpenVMS kit contains example programs that show you how to use the OpenSSL APIs in your OpenVMS application. This chapter includes a list of the example programs included in the kit, the program listings of two simple example programs, and SSL$EXAMPLES_SETUP.COM, which sets up the certificates and keys so you can run the example programs.

Programs Included in HP SSL Kit

When you install HP SSL for OpenVMS, the example programs are copied into SYS$COMMON:[SYSHLP.EXAMPLES.SSL]. The example programs included in the HP SSL kit are shown in Table 6-1.

Table 6-1 HP SSL Example Programs

Example Programs (Client and Server)	Description
SSL$SIMPLE_CLI.C and SSL$SIMPLE_SERV.C	Simple client/server programs. This client verifies the server certificate with the CA certificate. The client certificate is not loaded, and there is no client certificate verification in the SSL server.
SSL$BIO_CLI.C and SSL$BIO_SERV.C	Implement the same functionality as SSL$SIMPLE_CLI.C and SSL$SIMPLE_SERV.C by using socket BIOs.
SSL$CLI_VERIFY_CLIENT.C and SSL$SERV_VERIFY_CLIENT.C	Based on SSL$BIO_CLI.C and SSL$BIO_SERV.C. These programs perform the client certificate verification in the SSL server. For this purpose, the client certificate is loaded in the client, and the server has its CA certificate.
SSL$CLI_SESS_REUSE.C and SSL$SERV_SESS_REUSE.C	Demonstrate SSL session reuse (resumption). This feature was added to the implementation of SSL$BIO_CLI.C and BIO_SERV.C.
SSL$CLI_SESS_RENEGO.C and SSL$SERV_SESS_RENEGO.C	Demonstrate SSL renegotiation (rehandshake). This feature was added to the implementation of SSL$BIO_CLI.C and SSL$BIO_SERV.C.
SSL$CLI_SESS_REUSE_ CLI_VER.C and SSL$SERV_SESS_REUSE_ CLI_VER.C	Demonstrate SSL session reuse (resumption) as well as the client certificate verification in the server. The session reuse feature was added to the implementation of SSL$CLI_VERIFY_CLIENT.C and SSL$SERV_VERIFY_CLIENT.C.

Table 6-1 HP SSL Example Programs (Continued)

SSL$CLI_SESS_RENEGO_ CLI_VER.C and SSL$SERV_SESS_RENEGO_ CLI_VER.C	Demonstrate SSL renegotiation (rehandshake) as well as the client certificate verification. The renegotiation feature was added to the implementation of SSL$CLI_VERIFY_CLIENT.C and SSL$SERV_VERIFY_CLIENT.C.

Simple SSL Client Program

The following is the program listing of the SSL$SIMPLE_CLI.C example program.

```
/*
* ++
* FACILITY:
*
*Simplest SSL Client
*
* ABSTRACT:
*
*      This is an example of an SSL client with minimum functionality.
*      The socket APIs are used to handle TCP/IP operations.
*
*This SSL client verifies the server's certificate against the CA
*certificate loaded in the client.
*
*This SSL client does not load its own certificate and key because
*the SSL server does not request nor verify the client certificate.
*
*
* ENVIRONMENT:
*
*    OpenVMS Alpha V7.2-2
*    TCP/IP Services V5.0A or higher
*
* CREATION DATE:
*
*    1-Jan-2002
*
* --
*/
/* Assumptions, Build, Configuration, and Execution Instructions */
/*
*  ASSUMPTIONS:
*
*    The following are assumed to be true for the
*    execution of this program to succeed:
*
*    - SSL is installed and started on this system.
*
*    - this server program, and its accompanying client
*      program are run on the same system, but in different
```

```
*      processes.
*
*    - the certificate and keys referenced by this program
*      reside in the same directory as this program.  There
*      is a command procedure, SSL$EXAMPLES_SETUP.COM, to
*      help set up the certificates and keys.
*
*
*  BUILD INSTRUCTIONS:
*
*    To build this example program use commands of the form,
*
*      For a 32-bit application using only SSL APIs needs to run the
*      following commands for SSL_APP.C .
*       -----------------------------------------------------------------
*       $CC/POINTER_SIZE=32/PREFIX_LIBRARY_ENTRIES=ALL_ENTRIES SSL_APP.C
*       $LINK SSL_APP.OBJ, VMS_DECC_OPTIONS.OPT/OPT
*       -----------------------------------------------------------------
*       VMS_DECC_OPTIONS.OPT should include the following lines.
*       -------------------------------------------------
*       SYS$LIBRARY:OPENSSL$LIBCRYPTO_SHR32.EXE/SHARE
*       SYS$LIBRARY:OPENSSL$LIBSSL_SHR32.EXE/SHARE
*       -------------------------------------------------
*
*       Creating a 64-bit application of SSL_APP.C should run the
*       following commands.
*       -----------------------------------------------------------------
*       $CC/POINTER_SIZE=64/PREFIX_LIBRARY_ENTRIES=ALL_ENTRIES SSL_APP.C
*       $LINK SSL_APP.OBJ, VMS_DECC_OPTIONS.OPT/OPT
*       -----------------------------------------------------------------
*       VMS_DECC_OPTIONS.OPT should include the following lines.
*       -------------------------------------------------
*       SYS$LIBRARY:OPENSSL$LIBCRYPTO_SHR.EXE/SHARE
*       SYS$LIBRARY:OPENSSL$LIBSSL_SHR.EXE/SHARE
*       -------------------------------------------------
*
*
* CONFIGURATION INSTRUCTIONS:
*
*
* RUN INSTRUCTIONS:
*
*    To run this example program:
*
*    1) Start the server program,
*
*       $ run server on this system
*
*    2) Start the client program on this same system,
*
*       $ run client
*
*/

#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <netdb.h>
```

```
#include <unistd.h>
#ifdef __VMS
#include <socket.h>
#include <inet.h>

#include <in.h>
#else
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#endif

#include <openssl/crypto.h>
#include <openssl/ssl.h>
#include <openssl/err.h>

#define RETURN_NULL(x) if ((x)==NULL) exit (1)
#define RETURN_ERR(err,s) if ((err)==-1) { perror(s); exit(1); }
#define RETURN_SSL(err) if ((err)==-1) { ERR_print_errors_fp(stderr); exit(1); }

static int verify_callback(int ok, X509_STORE_CTX *ctx);

#define RSA_CLIENT_CERT"client.crt"
#define RSA_CLIENT_KEY "client.key"

#define RSA_CLIENT_CA_CERT      "client_ca.crt"
#define RSA_CLIENT_CA_PATH      "sys$common:[syshlp.examples.ssl]"

#define ON      1
#define OFF     0

void main()
{
  int err;

  int verify_client = OFF; /* To verify a client certificate, set ON */
  int sock;
  struct sockaddr_in server_addr;
  char*str;
  char  buf [4096];
  char hello[80];

SSL_CTX *ctx;
        SSL     *ssl;
SSL_METHOD *meth;
X509    *server_cert;
        EVP_PKEY        *pkey;

short int s_port = 5555;
const char*s_ipaddr = "127.0.0.1";

/*----------------------------------------------------------*/
  printf ("Message to be sent to the SSL server: ");
  fgets (hello, 80, stdin);

/* Load encryption & hashing algorithms for the SSL program */
SSL_library_init();
```

```
/* Load the error strings for SSL & CRYPTO APIs */
SSL_load_error_strings();

/* Create an SSL_METHOD structure (choose an SSL/TLS protocol version) */
  meth = SSLv3_method();

/* Create an SSL_CTX structure */
  ctx = SSL_CTX_new(meth);

RETURN_NULL(ctx);
/*-----------------------------------------------------------*/
if(verify_client == ON)

{

/* Load the client certificate into the SSL_CTX structure */
if (SSL_CTX_use_certificate_file(ctx, RSA_CLIENT_CERT,

     SSL_FILETYPE_PEM) <= 0) {
                ERR_print_errors_fp(stderr);
                exit(1);
}

/* Load the private-key corresponding to the client certificate */
        if (SSL_CTX_use_PrivateKey_file(ctx, RSA_CLIENT_KEY,
          SSL_FILETYPE_PEM) <= 0) {
                ERR_print_errors_fp(stderr);
                exit(1);
        }

/* Check if the client certificate and private-key matches */
        if (!SSL_CTX_check_private_key(ctx)) {
                fprintf(stderr,"Private key does not match the
                  certificate public key\n");
                exit(1);
        }
}

/* Load the RSA CA certificate into the SSL_CTX structure */
/* This will allow this client to verify the server's     */
/* certificate.                                           */


if (!SSL_CTX_load_verify_locations(ctx, RSA_CLIENT_CA_CERT, NULL)) {
               ERR_print_errors_fp(stderr);
               exit(1);
}

        /* Set flag in context to require peer (server) certificate */
        /* verification */

        SSL_CTX_set_verify(ctx,SSL_VERIFY_PEER,NULL);

        SSL_CTX_set_verify_depth(ctx,1);
  /* ------------------------------------------------------------ */
  /* Set up a TCP socket */

  sock = socket (PF_INET, SOCK_STREAM, IPPROTO_TCP);
```

```
RETURN_ERR(sock, "socket");

  memset (&server_addr, '\0', sizeof(server_addr));
  server_addr.sin_family      = AF_INET;

server_addr.sin_port        = htons(s_port);       /* Server Port number */

  server_addr.sin_addr.s_addr = inet_addr(s_ipaddr); /* Server IP */

/* Establish a TCP/IP connection to the SSL client */

  err = connect(sock, (struct sockaddr*) &server_addr, sizeof(server_addr));

RETURN_ERR(err, "connect");
  /* ----------------------------------------------- */
  /* An SSL structure is created */

  ssl = SSL_new (ctx);

RETURN_NULL(ssl);

/* Assign the socket into the SSL structure (SSL and socket without BIO) */
  SSL_set_fd(ssl, sock);

/* Perform SSL Handshake on the SSL client */
err = SSL_connect(ssl);

RETURN_SSL(err);

/* Informational output (optional) */
  printf ("SSL connection using %s\n", SSL_get_cipher (ssl));

  /* Get the server's certificate (optional) */
  server_cert = SSL_get_peer_certificate (ssl);

if (server_cert != NULL)
        {
printf ("Server certificate:\n");

str = X509_NAME_oneline(X509_get_subject_name(server_cert),0,0);
RETURN_NULL(str);
printf ("\t subject: %s\n", str);
free (str);

str = X509_NAME_oneline(X509_get_issuer_name(server_cert),0,0);
RETURN_NULL(str);
printf ("\t issuer: %s\n", str);
free(str);

X509_free (server_cert);

}
        else
                printf("The SSL server does not have certificate.\n");

/*-------- DATA EXCHANGE - send message and receive reply. -------*/
/* Send data to the SSL server */
  err = SSL_write(ssl, hello, strlen(hello));
```

```
RETURN_SSL(err);

/* Receive data from the SSL server */
  err = SSL_read(ssl, buf, sizeof(buf)-1);

RETURN_SSL(err);
  buf[err] = '\0';
  printf ("Received %d chars:'%s'\n", err, buf);

        /*--------------- SSL closure ---------------*/
        /* Shutdown the client side of the SSL connection */

        err = SSL_shutdown(ssl);
        RETURN_SSL(err);

        /* Terminate communication on a socket */
        err = close(sock);

        RETURN_ERR(err, "close");

        /* Free the SSL structure */
        SSL_free(ssl);

        /* Free the SSL_CTX structure */
        SSL_CTX_free(ctx);
}
```

Simple SSL Server Program

The following is the program listing of the SSL$SIMPLE_SERV.C example program.

```
/*
 * ++
 * FACILITY:
 *
 *Simplest SSL Server
 *
 * ABSTRACT:
 *
 *This is an example of a SSL server with minimum functionality.
 *The socket APIs are used to handle TCP/IP operations. This SSL
 *server loads its own certificate and key, but it does not verify
 *the certificate of the SSL client.
 *
 * ENVIRONMENT:
 *
 *    OpenVMS Alpha V7.2-2 or higher
 *    TCP/IP Services V5.0A or higher
 *
 *
 * CREATION DATE:
 *
 *    1-Jan-2002
```

```
 *
 * --
 */
/* Assumptions, Build, Configuration, and Execution Instructions */
/*
 *  ASSUMPTIONS:
 *
 *    The following are assumed to be true for the
 *    execution of this program to succeed:
 *
 *    - SSL is installed and started on this system.
 *
 *    - this server program, and its accompanying client
 *      program are run on the same system, but in different
 *      processes.
 *
 *    - the certificate and keys referenced by this program
 *      reside in the same directory as this program.  There
 *      is a command procedure, SSL$EXAMPLES_SETUP.COM, to
 *      help set up the certificates and keys.
 *
 *
 *  BUILD INSTRUCTIONS:
 *
 *    To build this example program use commands of the form,
 *
 *      For a 32-bit application using only SSL APIs needs to run the
 *      following commands for SSL_APP.C .
 *       -----------------------------------------------------------------
 *       $CC/POINTER_SIZE=32/PREFIX_LIBRARY_ENTRIES=ALL_ENTRIES SSL_APP.C
 *       $LINK SSL_APP.OBJ, VMS_DECC_OPTIONS.OPT/OPT
 *       -----------------------------------------------------------------
 *       VMS_DECC_OPTIONS.OPT should include the following lines.
 *       -------------------------------------------------
 *       SYS$LIBRARY:OPENSSL$LIBCRYPTO_SHR32.EXE/SHARE
 *       SYS$LIBRARY:OPENSSL$LIBSSL_SHR32.EXE/SHARE
 *       -------------------------------------------------
 *
 *       Creating a 64-bit application of SSL_APP.C should run the
 *       following commands.
 *       -----------------------------------------------------------------
 *       $CC/POINTER_SIZE=64/PREFIX_LIBRARY_ENTRIES=ALL_ENTRIES SSL_APP.C
 *       $LINK SSL_APP.OBJ, VMS_DECC_OPTIONS.OPT/OPT
 *       -----------------------------------------------------------------
 *       VMS_DECC_OPTIONS.OPT should include the following lines.
 *       -------------------------------------------------
 *       SYS$LIBRARY:OPENSSL$LIBCRYPTO_SHR.EXE/SHARE
 *       SYS$LIBRARY:OPENSSL$LIBSSL_SHR.EXE/SHARE
 *       -------------------------------------------------
 *
 *
 * CONFIGURATION INSTRUCTIONS:
 *
 *
 * RUN INSTRUCTIONS:
 *
 *    To run this example program:
 *
```

```
 *    1) Start the server program,
 *
 *       $ run server
 *
 *    2) Start the client program on this same system,
 *
 *       $ run client
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <netdb.h>
#include <unistd.h>

#ifdef __VMS
#include <types.h>
#include <socket.h>
#include <in.h>
#include <inet.h>

#else
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#endif

#include <openssl/crypto.h>
#include <openssl/ssl.h>
#include <openssl/err.h>

#define RSA_SERVER_CERT "server.crt"
#define RSA_SERVER_KEY "server.key"

#define RSA_SERVER_CA_CERT"server_ca.crt"
#define RSA_SERVER_CA_PATH"sys$common:[syshlp.examples.ssl]"

#define ON 1
#define OFF 0

#define RETURN_NULL(x) if ((x)==NULL) exit(1)
#define RETURN_ERR(err,s) if ((err)==-1) { perror(s); exit(1); }
#define RETURN_SSL(err) if ((err)==-1) { ERR_print_errors_fp(stderr); exit(1); }

void main()
{
int err;
int verify_client = OFF; /* To verify a client certificate, set ON */

  int listen_sock;
  int sock;
  struct sockaddr_in sa_serv;
  struct sockaddr_in sa_cli;
  size_t client_len;
  char*str;
  char     buf[4096];
```

```
SSL_CTX*ctx;
        SSL*ssl;
  SSL_METHOD *meth;

X509*client_cert = NULL;

short int       s_port = 5555;
/*----------------------------------------------------------------*/
/* Load encryption & hashing algorithms for the SSL program */
SSL_library_init();

/* Load the error strings for SSL & CRYPTO APIs */
SSL_load_error_strings();

/* Create a SSL_METHOD structure (choose a SSL/TLS protocol version) */
  meth = SSLv3_method();

/* Create a SSL_CTX structure */
  ctx = SSL_CTX_new(meth);

if (!ctx) {

ERR_print_errors_fp(stderr);

exit(1);

}

/* Load the server certificate into the SSL_CTX structure */
if (SSL_CTX_use_certificate_file(ctx, RSA_SERVER_CERT, SSL_FILETYPE_PEM) <= 0) {

    ERR_print_errors_fp(stderr);

    exit(1);

}

/* Load the private-key corresponding to the server certificate */
  if (SSL_CTX_use_PrivateKey_file(ctx, RSA_SERVER_KEY, SSL_FILETYPE_PEM) <= 0) {

    ERR_print_errors_fp(stderr);
    exit(1);
  }

/* Check if the server certificate and private-key matches */
if (!SSL_CTX_check_private_key(ctx)) {

    fprintf(stderr,"Private key does not match the certificate public key\n");
    exit(1);
  }

if(verify_client == ON)

{

/* Load the RSA CA certificate into the SSL_CTX structure */
if (!SSL_CTX_load_verify_locations(ctx, RSA_SERVER_CA_CERT, NULL)) {
```

```
                ERR_print_errors_fp(stderr);
                exit(1);
        }

/* Set to require peer (client) certificate verification */
SSL_CTX_set_verify(ctx,SSL_VERIFY_PEER,NULL);

/* Set the verification depth to 1 */
SSL_CTX_set_verify_depth(ctx,1);

}
/* ----------------------------------------------- */
/* Set up a TCP socket */

listen_sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);

RETURN_ERR(listen_sock, "socket");
  memset (&sa_serv, '\0', sizeof(sa_serv));
  sa_serv.sin_family      = AF_INET;
  sa_serv.sin_addr.s_addr = INADDR_ANY;
  sa_serv.sin_port        = htons (s_port);          /* Server Port number */
  err = bind(listen_sock, (struct sockaddr*)&sa_serv,sizeof(sa_serv));

RETURN_ERR(err, "bind");

  /* Wait for an incoming TCP connection. */
  err = listen(listen_sock, 5);

RETURN_ERR(err, "listen");
  client_len = sizeof(sa_cli);

/* Socket for a TCP/IP connection is created */
  sock = accept(listen_sock, (struct sockaddr*)&sa_cli, &client_len);

  RETURN_ERR(sock, "accept");
  close (listen_sock);

  printf ("Connection from %lx, port %x\n", sa_cli.sin_addr.s_addr,
   sa_cli.sin_port);

/* ----------------------------------------------- */
/* TCP connection is ready. */
/* A SSL structure is created */
ssl = SSL_new(ctx);

RETURN_NULL(ssl);

/* Assign the socket into the SSL structure (SSL and socket without BIO) */
SSL_set_fd(ssl, sock);

/* Perform SSL Handshake on the SSL server */
err = SSL_accept(ssl);

RETURN_SSL(err);

  /* Informational output (optional) */
  printf("SSL connection using %s\n", SSL_get_cipher (ssl));
```

```
if (verify_client == ON)
{

    /* Get the client's certificate (optional) */
    client_cert = SSL_get_peer_certificate(ssl);
    if (client_cert != NULL)
    {

    printf ("Client certificate:\n");
    str = X509_NAME_oneline(X509_get_subject_name(client_cert), 0, 0);
    RETURN_NULL(str);
    printf ("\t subject: %s\n", str);
    free (str);
    str = X509_NAME_oneline(X509_get_issuer_name(client_cert), 0, 0);
    RETURN_NULL(str);
    printf ("\t issuer: %s\n", str);
    free (str);
    X509_free(client_cert);
    }

    else

    printf("The SSL client does not have certificate.\n");
}

/*------- DATA EXCHANGE - Receive message and send reply. -------*/
/* Receive data from the SSL client */
  err = SSL_read(ssl, buf, sizeof(buf) - 1);

RETURN_SSL(err);

  buf[err] = '\0';

  printf ("Received %d chars:'%s'\n", err, buf);

/* Send data to the SSL client */
  err = SSL_write(ssl, "This message is from the SSL server",

   strlen("This message is from the SSL server"));

RETURN_SSL(err);

/*--------------- SSL closure ---------------*/
/* Shutdown this side (server) of the connection. */

err = SSL_shutdown(ssl);

RETURN_SSL(err);

/* Terminate communication on a socket */
err = close(sock);

RETURN_ERR(err, "close");

/* Free the SSL structure */
SSL_free(ssl);
```

```
/* Free the SSL_CTX structure */
 SSL_CTX_free(ctx);

}
```

Creating Certificates and Keys for the Example Programs

The command procedure SSL$EXAMPLES_SETUP.TEMPLATE (located in SYS$COMMON:[SYSHLP.EXAMPLES.SSL]) is a template that sets up the certificate and keys so you can run the example programs shown in the previous sections. SSL$EXAMPLES_SETUP.TEMPLATE does the following:

- Creates a Certificate Authority (CA) certificate
- Creates server and client certificate requests
- The CA signs the two certificate requests
- Creates server and client certificates

To execute this command procedure, be sure that SSL$STARTUP.COM and SSL$UTILS.COM have been run, then remove the comment characters from the commands.

The following program listing shows SSL$EXAMPLES_SETUP.TEMPLATE.

```
$!
$!  SSL$EXAMPLES_SETUP.COM --
$!
$! This command procedure is actually a template that will show
$! the commands necessary to create certificates and keys for the example
$! programs.
$!
$! Also included in this file are the necessary options to enter into the
$! SSL$CERT_TOOL.COM to create the necessary certificates and keys to the
$! example programs.  The SSL$CERT_TOOL.COM is found in SSL$COM.  See the
$! documenation for more information about the SSL$CERT_TOOL.COM.
$!
$! 1. Create CA certificate - option 5 in SSL$CERT_TOOL.COM.
$!    This will create a key in one file, named SSL$KEY:SERVER_CA.KEY
$!    by default, and a certificate in another file, named
$!    SSL$CRT:SERVER_CA.CRT by default.
$!
$! 2. Make 2 copies of CA certificate created in step #1.
$!    One should be called server_ca.crt and the other called
$!    client_ca.crt as these are the filenames defined in the
$!    example programs.  You will have to exit the SSL$CERT_TOOL.COM
$!    procedure to do this operation from the DCL command line.
$!    For example:
$!$ COPY SSL$KEY:SERVER_CA.KEY SSL$KEY:CLIENT_CA.KEY
$!$ COPY SSL$CRT:SERVER_CA.CRT SSL$CRT:CLIENT_CA.CRT
$!
$! 3. Create a server certificate signing request - option 3 in SSL$CERT_TOOL.COM.
$!    The Common Name should be the TCP/IP hostname of the server system.
$!    The default name of the request is SERVER.CSR.  The corresponding private
$!    key is named SERVER.KEY.
```

```
$!
$! 4. Sign server certificate signing request - option 6 in SSL$CERT_TOOL.COM
$!    Use the CA certificate, SERVER_CA.CRT, created in step #1 to sign the request
$!    created in step #3.  This will create a certificate file, which should be
$!    named SERVER.CRT.  This is the name as it is defined in example programs.
$!
$! 5. Create a client certificate signing request - option 3 in SSL$CERT_TOOL.COM.
$!
$! 6. Sign client certificate signing request - option 6 in SSL$CERT_TOOL.COM
$!    Use the CA certificate, CLIENT_CA.CRT, created in step #1 to sign the request
$!    created in step #5.  This will create a certificate file, which should be
$!    named CLIENT.CRT.  This is the name as it is defined in example programs.
$!
$! 7. These certificates and keys should reside in the same directory as
$!    the example programs.
$!
$! The commands have been changed to use generic data as
$! input.  To use these commands, one will have to substitute
$! the generic data with data specific to their site.
$! For example, yourcountry could be change to US.  It is
$! assumed that the SSL startup file, SYS$STARTUP:SSL$STARTUP.COM,
$! and the SSL$COM:SSL$UTILS.COM procedures have been executed.
$!
$! Set up some random data.
$!
$! $ show system/full/output=randfile.
$!
$!
$! Check to make sure the SERIAL and INDEX files exist.
$! If they don't, create them.
$!
$! $ if f$search ("SSL$PRIVATE:SERIAL.TXT") .eqs. ""
$! $ then
$! $   CREATE SSL$PRIVATE:SERIAL.TXT
$! 01
$! $ endif
$!
$! $ if f$search ("SSL$PRIVATE:INDEX.TXT") .eqs. ""
$! $ then
$! $   CREATE SSL$PRIVATE:INDEX.TXT
$! $ endif
$!
$! Create the CA certificate.
$!
$! $ define/user sys$command sys$input
$! $ openssl req -config ssl$root:[000000]openssl-vms.cnf -new -x509
    -days 1825 -keyout ca.key -out ca.crt
$! yourpassword
$! yourpassword
$! yourcountry
$! yourstate
$! yourcity
$! yourcompany
$! yourdepartment
$! your Certificate Authority certificate
$! firstname.lastname@yourcompany.com
$! $!
$! $!
```

```
$! $! Create the server certificate request.
$! $!
$! $!   Note : There is no way to use the value of a
$! $!          symbol when you are using the value of
$! $!          symbol as input, as we do below.  To get
$! $!          around, we create a .COM on the fly and
$! $!          execute the created .COM file to create
$! $!          the server certificate.
$! $!
$! $ hostname = f$trnlnm("tcpip$inet_host")
$! $ domain = f$trnlnm("tcpip$inet_domain")
$! $ server_name = hostname + "." + domain"
$! $!
$! $ open/write s_com create_s_cert.com
$! $!
$! $ write s_com "$!"
$! $ write s_com "$ define/user sys$command sys$input
$! $ write s_com "$ openssl req -new -nodes -config
     ssl$root:[000000]openssl-vms.cnf -keyout server.key -out server.csr"
$! $ write s_com "yourcountry"
$! $ write s_com "yourstate"
$! $ write s_com "yourcity"
$! $ write s_com "yourcompany"
$! $ write s_com "yourdepartment"
$! $ write s_com "''server_name'"
$! $ write s_com "firstname.lastname@yourcompany.com"
$! $ write s_com ""
$! $ write s_com ""
$! $!
$! $ close s_com
$! $ @create_s_cert
$! $ delete create_s_cert.com;
$! $!
$! $!
$! $! Now, sign the server certificate ...
$! $!
$! $ define/user sys$command sys$input
$! $ openssl ca -config ssl$root:[000000]openssl-vms.cnf -cert
     ca.crt -keyfile ca.key -out server.crt -infiles server.csr
$! yourpassword
$! Y
$! Y
$! $!
$! $!
$! $! Create the client certificate request.
$! $!
$! $ define/user sys$command sys$input
$! $ openssl req -new -nodes -config ssl$root:[000000]openssl-vms.cnf
     -keyout client.key -out client.csr
$! yourcountry
$! yourstate
$! yourcity
$! yourcompany
$! yourdepartment
$! yourname
$! firstname.lastname@yourcompany.com
$!
$!
```

```
$! $!
$! $!
$! $! Now, sign the client certificate ...
$! $!
$! $ define/user sys$command sys$input
$! $ openssl ca -config ssl$root:[000000]openssl-vms.cnf -cert
     ca.crt -keyfile ca.key -out client.crt -infiles client.csr
$! yourpassword
$! Y
$! Y
$! $!
$! $! Let's view the CA certificate.
$! $!
$! $ openssl x509 -noout -text -in ca.crt
$! $!
$! $!
$! $! Let's view the Server Certificate Request.
$! $!
$! $ openssl req -noout -text -in server.csr
$! $!
$! $! Let's view the Server Certificate.
$! $!
$! $ openssl x509 -noout -text -in server.crt
$! $!
$! $! Let's view the Client Certificate Request.
$! $!
$! $ openssl req -noout -text -in client.csr
$! $!
$! $! Let's view the Client Certificate.
$! $!
$! $ openssl x509 -noout -text -in client.crt
$! $!
$! $!
$! $exit
```

CRYPTO and SSL Application Programming Interface (API) Reference

This reference section includes the OpenSSL **Crypto** and **SSL** APIs, and is based on information provided by The Open Group. This information can also be found at the following URL:

`http://www.openssl.org`

Crypto APIs

The OpenSSL Crypto library implements a wide range of cryptographic algorithms used in various Internet standards. The services provided by this library are used by the OpenSSL implementations of SSL, TLS and S/MIME, and they have also been used to implement SSH, OpenPGP, and other cryptographic standards. The Crypto library consists of a number of sublibraries that implement the individual algorithms. The functionality includes symmetric encryption, public key cryptography and key agreement, certificate handling, cryptographic hash functions and a cryptographic pseudorandom number generator.

The Crypto library is provided in the form of a shareable image and is located at:

SYS$LIBRARY:SSL$LIBCRYPTO_SHR.EXE (for 64-bit APIs)
SYS$LIBRARY:SSL$LIBCRYPTO_SHR32.EXE (for 32-bit APIs)

NOTE The documentation for the following Crypto APIs are not included in this manual. The APIs themselves are provided in the HP SSL for OpenVMS kit and can be found in the preceding shareable images.

```
X509_NAME_oneline()
X509_STORE_CTX_get_current_cert()
X509_STORE_CTX_get_error()
X509_STORE_CTX_get_error_depth()
X509_STORE_CTX_get_ex_data()
X509_STORE_CTX_set_error()
X509_verify_cert_error_string()
X509_get_issuer_name()
X509_get_pubkey()
X509_get_subject_name()
X509_free()
```

SSL APIs

The OpenSSL SSL library implements the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) protocols.

This library is provided in the form of a shareable image and is located at:

SYS$LIBRARY:SSL$LIBSSL_SHR.EXE (for 64-bit APIs)
SYS$LIBRARY:SSL$LIBSSL_SHR32.EXE (for 32-bit APIs)

The C header files (.H) that contain the prototypes for these APIs are found in SSL$ROOT:[INCLUDE]. A logical name, SSL$INCLUDE, allows you to access this directory. The logical name OPENSSL, which points to SSL$INCLUDE, is provided so that applications can use statements similar to the following:

```
#include <openssl/include.filename.h>
```

NOTE Do not confuse the OPENSSL logical name with the OPENSSL foreign symbol. The foreign symbol provides access to the OpenSSL command line interface.

Table 1 lists all of the APIs included in HP SSL for OpenVMS and categorizes the APIs by function.

Table 1 HP SSL APIs Grouped by Function

CA Perl Script for Interface to OpenSSL Certificate Programs	
CA.pl	
OpenSSL Utility Commands	
asn1parse	pkcs7
ca	pkcs8
ciphers	rand
crl	req
crl2pkcs7	rsa
dgst	rsautl
dHParam	s_client
dsa	s_server
dsaparam	sess_id
enc	smime
gendsa	speed
genrsa	spkac
nseq	verify
openssl	version
passwd	x509
pkcs12	
BIO Crypto Sublibrary — Provides I/O Abstraction APIs	
bio	BIO_push
BIO_ctrl	BIO_read
BIO_ctrl_get_read_request	BIO_s_accept
BIO_ctrl_pending	BIO_s_bio
BIO_f_base64	BIO_s_connect
BIO_f_buffer	BIO_s_fd
BIO_f_cipher	BIO_s_file
BIO_f_md	BIO_s_mem
BIO_f_null	BIO_s_null
BIO_f_ssl	BIO_s_socket

Table 1 HP SSL APIs Grouped by Function (Continued)

BIO_find_type	BIO_set_callback
BIO_new	BIO_should_retry
BIO_new_bio_pair	
BIGNUM Crypto Sublibrary — Provides Multiprecision Integer Arithmetic APIs	
bn	bn_internal
BN_CTX_new	BN_mod_inverse
BN_CTX_start	BN_mod_mul_montgomery
BN_add	BN_mod_mul_reciprocal
BN_add_word	BN_new
BN_bn2bin	BN_num_bytes
BN_cmp	BN_rand
BN_copy	BN_set_bit
BN_generate_prime	BN_zero
Blowfish Crypto Sublibrary — Provides the Blowfish Cipher APIs	
blowfish	
Buffer Crypto Sublibrary — Provides APIs to Manipulate Simple Character Arrays	
buffer	
OpenSSL Cryptographic Library	
crypto	CRYPTO_set_ex_data
d2i Sublibrary	
d2i_DHparamS	d2i_SSL_SESSION
d2i_RSAPublicKey	
DES Crypto Sublibrary — Provides DES Encryption Algorithm APIs	
des	
DH Crypto Sublibrary — Provides Diffie-Hellman Key Agreement APIs	
dh	DH_new
DH_generate_key	DH_set_method
DH_generate_parameters	DH_size
DH_get_ex_new_index	
DSA Crypto Sublibrary — Provides Digital Signature Algorithm (DSA) APIs	
dsa	DSA_get_ex_new_index

Table 1 HP SSL APIs Grouped by Function (Continued)

DSA_SIG_new	DSA_new
DSA_do_sign	DSA_set_method
DSA_dup_DH	DSA_sign
DSA_generate_key	DSA_size
DSA_generate_parameters	
ERR Crypto Sublibrary — Provides Error Handling APIs	
err	ERR_load_crypto_strings
ERR_GET_LIB	ERR_load_strings
ERR_clear_error	ERR_print_errors
ERR_error_string	ERR_put_error
ERR_get_error	RR_remove_state
ERR_load_SSL_strings	
EVP Crypto Sublibrary — Provides High-Level Cryptographic APIs	
evp	EVP_SealInit
EVP_DigestInit	EVP_SignInit
EVP_EncryptInit	EVP_VerifyInit
EVP_OpenInit	
HMAC Crypto Sublibrary — Provides Hashed Message Authentication Code API	
HMAC	
MDC2 Crypto Sublibrary — Provides Message Digest API with DES Algorithm	
MDC2	
LH_Stats Crypto Sublibrary — Provides API to Access the Lhash Structure Stats	
lh_stats	
LHASH Crypto Sublibrary — Provides API to Implement Dynamic Hash Tables	
lhash	
MD5 Crypto Sublibrary — Provides MD5 Cryptographic Hash API	
md5	
Miscellaneous OpenSSL APIs	
OPENSSL_VERSION_NUMBER	OpenSSL_add_all_algorithms
PEM Crypto Sublibrary — Provides APIs to Read and Write Data in a PEM Format	
pem	

Table 1 HP SSL APIs Grouped by Function (Continued)

RAND Crypto Sublibrary — Provides Pseudo-Random Number Generator APIs	
rand_ssl	RAND_egd
RAND_add	RAND_load_file
RAND_bytes	RAND_set_rand_method
RAND_cleanup	
RIPEMD160 Crypto Sublibrary — Provides Cryptographic Hash Function with 160-Bit Output	
RIPEMD160	
RC4 Crypto Sublibrary — Provides RC4 Encryption APIs	
rc4	
RSA Crypto Sublibrary — Provides RSA Public Key Cryptosystem APIs	
rsa	RSA_print
RSA_blinding_on	RSA_private_encrypt
RSA_check_key	RSA_public_encrypt
RSA_generate_key	RSA_set_method
RSA_get_ex_new_index	RSA_sign
RSA_new	RSA_sign_ASN1_OCTET_STRING
RSA_padding_add_PKCS1_type_1	RSA_size
SHA Crypto Sublibrary — Provides Secure Hash Algorithm (SHA-1) APIs	
sha	
Threads Crypto Sublibrary — Provides APIS to Implement Thread-Safe Code	
threads	
OpenSSL CONF Library Configuration Files	
config	
Variants of DES and Other Crypto Algorithms of OpenSSL	
des_modes	
SSL_CTX Data Structure APIs	
SSL_CTX_add_extra_chain_cert	SSL_CTX_set_client_CA_list
SSL_CTX_add_session	SSL_CTX_set_def_verify_paths
SSL_CTX_ctrl	SSL_CTX_set_default_passwd_cb
SSL_CTX_flush_sessions	SSL_CTX_set_info_callback
SSL_CTX_free	SL_CTX_set_mode

Table 1 HP SSL APIs Grouped by Function (Continued)

SSL_CTX_get_cert_store	SSL_CTX_set_options
SSL_CTX_get_ex_new_index	SSL_CTX_set_purpose
SSL_CTX_get_quiet_shutdown	SSL_CTX_set_quiet_shutdown
SSL_CTX_get_verify_mode	SSL_CTX_set_session_cache_mode
SSL_CTX_load_verify _locations	SSL_CTX_set_session_id_context
SSL_CTX_new	SSL_CTX_set_ssl_version
SSL_CTX_sess_number	SL_CTX_set_timeout
SSL_CTX_sess_set_cache_size	SSL_CTX_set_tmp_dh_callback
SSL_CTX_sess_set_get_cb	SSL_CTX_set_tmp_rsa_callback
SSL_CTX_sessions	SSL_CTX_set_trust
SSL_CTX_set_cert_store	SSL_CTX_set_verify
SSL_CTX_set_cert_verify_cb	SSL_CTX_use_certificate
SSL_CTX_set_cipher_list	
SSL_SESSION Data Structure APIs	
SSL_SESSION_cmp	SSL_SESSION_hash
SSL_SESSION_free	SSL_SESSION_new
SSL_SESSION_get_ex_new_index	SSL_SESSION_print
SSL_SESSION_get_time	
SSL APIs	
ssl	SSL_get_rbio
SSL_accept	SSL_get_re ad_ahead
SSL_alert_desc_string.	SL_get_session
SSL_alert_type_string	SSL_get_shared_ciphers
SSL_callback_ctrl	SSL_get_verify_result
SSL_check_private_key	SSL_get_version
SSL_CIPHER_get_name	SSL_library_init
SSL_clear	SSL_load_client_CA_file
SSL_COMP_add_compression_method	SSL_new
SSL_connect	SL_peek
SSL_copy_session_id	SSL_pending
SSL_ctrl	SSL_read

Table 1 HP SSL APIs Grouped by Function (Continued)

SSL_do_handshake	SSL_renegotiate
SSL_dup	SSL_rstate_string
SSL_dup_CA_list	SSL_session_reused
SSL_free	SSL_set_bio
SSL_get_SSL_CTX	SSL_set_connect_state
SSL_get_certificate	SSL_set_fd
SSL_get_ciphers	SSL_set_info_callback
SSL_get_client _CA_list	SSL_set_purpose
SSL_get_current_cipher	SSL_set_quiet_shutdown
SSL_get_default_timeout	SSL_set_read_ahead
SSL_get_error	SSL_set_session
SSL_get_ex_data_ X509_STORE_CTX_idx	SSL_set_shutdown
SSL_get_ex_new_index	SSL_set_trust
SSL_get_fd	SSL_set_verify_result
SSL_get_finished	SSL_shutdown
SSL_get_info_callback	SSL_state
SSL_get_peer_cert_chain	SSL_state_string
SSL_get_peer_certificate	SSL_version
SSL_get_peer_finished	SSL_want
SSL_get_privatekey	SSL_write
SSL_get_quiet_shutdown	

asn1parse

NAME

asn1parse – ASN.1 parsing tool

SYNOPSIS

`openssl asn1parse [-inform PEM|DER] [- in` *filename*`] [-out` *filename*`] [-noout] [-offset` *number*`] [-length` *number*`] [-i] [-oid` *filename*`] [-strparse` *offset*`]`

OPTIONS

`inform DER|PEM`

The input format. `DER` is binary format and `PEM` (the default) is base64 encoded.

`in` *filename*

The input file, default is standard input

`out` *filename*

Output file to place the DER encoded data into. If this option is not present then no data will be output. This is most useful when combined with the `strparse` option.

`noout`

Does not output the parsed version of the input file.

`offset` *number*

Starting offset to begin parsing, default is start of file.

`length` *number*

Number of bytes to parse, default is until end of file.

`i`

Indents the output according to the depth of the structures.

`oid` *filename*

A file containing additional object identifiers (OIDs). The format of this file is described in the Notes section below.

`strparse` *offset*

Parses the content octets of the ASN.1 object starting at `offset`. This option can be used multiple times to drill down into a nested structure.

DESCRIPTION

The `asn1parse` command is a diagnostic utility that can parse ASN.1 structures. It can also be used to extract data from ASN.1 formatted data.

NOTES

If an OID is not part of OpenSSL's internal table it will be represented in numerical form (for example 1.2.3.4). The file passed to the `oid` option allows additional OIDs to be included. Each line consists of three columns, the first column is the OID in numerical format and should be followed by white space. The second column is the short name which is a single word followed by white space. The final column is the rest of the line and is the long name. `asn1parse` displays the long name. For example:

```
1.2.3.4shortNameA long name
```

RESTRICTIONS

There should be options to change the format of input lines. The output of some ASN.1 types is not handled well.

EXAMPLES

The output will typically contain lines like these:

```
  0:d=0  hl=4 l= 681 cons: SEQUENCE
  .....
  229:d=3  hl=3 l= 141 prim: BIT STRING
  373:d=2  hl=3 l= 162 cons: cont [ 3 ]
  376:d=3  hl=3 l= 159 cons: SEQUENCE
  379:d=4  hl=2 l=  29 cons: SEQUENCE
  381:d=5  hl=2 l=   3 prim: OBJECT            :X509v3 Subject Key Identifier
  386:d=5  hl=2 l=  22 prim: OCTET STRING
  410:d=4  hl=2 l= 112 cons: SEQUENCE
  412:d=5  hl=2 l=   3 prim: OBJECT            :X509v3 Authority Key Identifier
  417:d=5  hl=2 l= 105 prim: OCTET STRING
  524:d=4  hl=2 l=  12 cons: SEQUENCE
  .....
```

This example is part of a self-signed certificate. Each line starts with the offset in decimal. The `d=XX` specifies the current depth. The depth is increased within the scope of any SET or SEQUENCE. The `hl=XX` gives the header length (tag and length octets) of the current type. The `l=XX` gives the length of the contents octets.

The `i` option can be used to make the output more readable.

Some knowledge of the ASN.1 structure is needed to interpret the output.

In this example the bit string at offset 229 is the certificate public key. The content octets of this will contain the public key information. This can be examined by using the option `strparse 229` to yield:

```
0:d=0  hl=3 l= 137 cons: SEQUENCE
3:d=1  hl=3 l= 129 prim: INTEGER   :E5D21E1F5C8D208EA7A2166C7FAF9F6BDF2059669C60876DDB70840
F1A5AAFA59699FE471F379F1DD6A487E7D5409AB6A88D4A9746E24B91
D8CF55DB3521015460C8EDE44EE8A4189F7A7BE77D6CD3A9AF2696F486855
CF58BF0EDF2B4068058C7A947F52548DDF7E15E96B385F86422BEA9064A3
EE9E1158A56E4A6F47E5897
135:d=1  hl=2 l=   3 prim: INTEGER        :010001
```

bio

NAME

bio – I/O abstraction

SYNOPSIS

```
#include <openssl/bio.h>
```

DESCRIPTION

A BIO is an I/O abstraction. It hides many of the underlying I/O details from an application. If an application uses a BIO for its I/O it can transparently handle SSL connections, unencrypted network connections and file I/O.

There are two types of BIO, a source/sink BIO and a filter BIO.

As its name implies a source/sink BIO is a source and/or sink of data, examples include a socket BIO and a file BIO.

A filter BIO takes data from one BIO and passes it through to another, or the application. The data may be left unmodified (for example, a message digest BIO) or translated (for example, an encryption BIO). The effect of a filter BIO may change according to the I/O operation it is performing. For example, an encryption BIO will encrypt data if it is being written to and will decrypt data if it is being read from.

BIOs can be joined to form a chain (a single BIO is a chain with one component). A chain normally consist of one source/sink BIO and one or more filter BIOs. Data read from or written to the first BIO then traverses the chain to the end (usually a source/sink BIO).

SEE ALSO

Functions: *BIO_ctrl*, *BIO_f_base64*, *BIO_f_buffer*, *BIO_f_cipher*, *BIO_f_md*, *BIO_f_null*, *BIO_f_ssl*, *BIO_find_type*, *BIO_new*, *BIO_new_bio_pair*, *BIO_push*, *BIO_read*, *BIO_s_accept*, *BIO_s_bio*, *BIO_s_connect*, *BIO_s_fd*, *BIO_s_file*, *BIO_s_mem*, *BIO_s_null*, *BIO_s_socket*, *BIO_set_callback*, *BIO_should_retry*

BIO_ctrl

NAME

BIO_ctrlBIO_callback_ctrl, BIO_ctrl_pending, BIO_ctrl_wpending, BIO_wpending, BIO_eof, BIO_flushBIO_get_close, BIO_get_info_callback, BIO_set_info_callback, BIO_int_ctrl, BIO_pending, BIO_ptr_ctrl, BIO_reset, BIO_seek, BIO_set_close, BIO_tell – BIO control operations

SYNOPSIS

```
#include <openssl/bio.h>

long BIO_ctrl(
        BIO *bp,int cmd,long larg,void *parg
);

long BIO_callback_ctrl(
        BIO *b, int cmd, void (*fp)(struct bio_st *, int, const char *, int, long, long)
);

char *BIO_ptr_ctrl(
        BIO *bp,int cmd,long larg
);

long BIO_int_ctrl(
        BIO *bp,int cmd,long larg,int iarg
);

int BIO_reset(
        BIO *b
);

int BIO_seek(
        BIO *b, int ofs
);

int BIO_tell(
        BIO *b
);

int BIO_flush(
        BIO *b
);

int BIO_eof(
        BIO *b
);

int BIO_set_close(
        BIO *b,long flag
);

int BIO_get_close(
```

```
        BIO *b
);
int BIO_pending(
        BIO *b
);
int BIO_wpending(
        BIO *b
);
size_t BIO_ctrl_pending(
        BIO *b
);
size_t BIO_ctrl_wpending(
        BIO *b
);
int BIO_get_info_callback(
        BIO *b,bio_info_cb **cbp
);
int BIO_set_info_callback(
        BIO *b,bio_info_cb *cb
);
typedef void bio_info_cb(
        BIO *b, int oper, const char *ptr, int arg1, long arg2, long arg3
);
```

DESCRIPTION

The `BIO_ctrl()`, `BIO_callback_ctrl()`, `BIO_ptr_ctrl()`, and `BIO_int_ctrl()` functions are BIO control operations taking arguments of various types. These functions are not usually called directly; various macros are used instead. The standard macros are described below. Macros specific to a particular type of BIO are described in the specific BIO's reference page, as well as any special features of the standard calls.

The `BIO_reset()`function typically resets a BIO to some initial state. In the case of file related BIOs, for example, it rewinds the file pointer to the start of the file.

The `BIO_seek()` function resets a file related BIO's (its file descriptor and FILE BIOs) file position pointer to `ofs` bytes from start of file.

The `BIO_tell()` function returns the current file position of a file related BIO.

The `BIO_flush()` function normally writes out any internally buffered data, in some cases it is used to signal EOF and that no more data will be written.

The `BIO_eof()` function returns 1 if the BIO has read EOF, the precise meaning of EOF varies according to the BIO type.

The `BIO_set_close()` function sets the BIO `b` close flag to `flag`. `flag` can take the value `BIO_CLOSE` or `BIO_NOCLOSE`. Typically `BIO_CLOSE` is used in a source/sink BIO to indicate that the underlying I/O stream should be closed when the BIO is freed.

The BIO_get_close() function returns the BIO's close flag.

The BIO_pending(), BIO_ctrl_pending(), BIO_wpending(), and BIO_ctrl_wpending() functions return the number of pending characters in the BIO's read and write buffers. Not all BIOs support these calls. The BIO_ctrl_pending() and BIO_ctrl_wpending() functions return a size_t type and are functions. BIO_pending() and BIO_wpending() are macros which call BIO_ctrl().

NOTES

The BIO_flush() function, because it can write data, might return 0 or -1 indicating that the call should be retried later in a similar manner to BIO_write(). The BIO_should_retry() function should be used and appropriate action taken if the call fails.

The return values of the BIO_pending() and BIO_wpending() functions might not reliably determine the amount of pending data in all cases. For example, in the case of a file BIO some data may be available in the FILE structure's internal buffers but it is not possible to determine this in a portably way. For other types of BIO they might not be supported.

Filter BIOs, if they do not internally handle a particular BIO_ctrl() operation, usually pass the operation to the next BIO in the chain. This often means there is no need to locate the required BIO for a particular operation. It can be called on a chain and it will be automatically passed to the relevant BIO. However, this can cause unexpected results. For example, no current filter BIOs implement BIO_seek(), but this might still succeed if the chain ends in a FILE or file descriptor BIO.

Source/sink BIOs return a 0 if they do not recognize the BIO_ctrl() operation.

RESTRICTIONS

Some of the return values are ambiguous and care should be taken. In particular a return value of 0 can be returned if an operation is not supported, if an error occurred, if EOF has not been reached, and in the case of BIO_seek() - on a file BIO for a successful operation.

RETURN VALUES

BIO_reset() normally returns 1 for success and 0 or -1 for failure. File BIOs are an exception, they return 0 for success and -1 for failure.

BIO_seek() and BIO_tell() both return the current file position on success and -1 for failure, except file BIOs which for BIO_seek() always return 0 for success and -1 for failure.

BIO_flush() returns 1 for success and 0 or -1 for failure.

BIO_eof() returns 1 if EOF has been reached 0 otherwise.

BIO_set_close() always returns 1.

BIO_get_close() returns the close flag value: BIO_CLOSE or BIO_NOCLOSE.

The BIO_pending(), BIO_ctrl_pending(), BIO_wpending(), and BIO_ctrl_wpending() functions return the amount of pending data.

BIO_ctrl_get_read_request

NAME

BIO_ctrl_get_read_request – Find out how many bytes were requested from the BIO

SYNOPSIS

```
#include <openssl/bio.h>
size_t BIO_ctrl_get_read_request(
        BIO *bio
);
```

DESCRIPTION

The `BIO_ctrl_get_read_request()` function returns the number of bytes that were last requested from `bio` by a `BIO_read()` operation. This is useful for BIO pairs, for example, so that the application knows how many bytes to supply to bio.

NOTES

When bio is NULL, the OpenSSL library calls `assert()`.

RETURN VALUES

The following return values can occur:

>=0

The number of bytes requested.

SEE ALSO

Functions: *bio*, *BIO_s_mem*, *BIO_new_bio_pair*

BIO_ctrl_pending

NAME

BIO_ctrl_pending – Find out how many bytes are buffered in a BIO

SYNOPSIS

```
#include <openssl/bio.h>
size_t BIO_ctrl_pending(
        BIO *bio
);
```

DESCRIPTION

The `BIO_ctrl_pending()` function returns the number of bytes buffered in a BIO.

NOTES

When bio is NULL, the OpenSSL library calls `assert()`.

RETURN VALUES

The following return values can occur:

>=0

The number of bytes pending the BIO.

SEE ALSO

Functions: *bio*, *BIO_s_mem*, *BIO_new_bio_pair*

BIO_f_base64

NAME

BIO_f_base64 – BIO filter for base64

SYNOPSIS

```
#include <openssl/bio.h>
#include <openssl/evp.h>

BIO_METHOD *BIO_f_base64(
       void
);
```

DESCRIPTION

The `BIO_f_base64()` function returns the base64 BIO method. This is a filter BIO that base64 encodes any data written through it and decodes any data read through it.

Base64 BIOs do not support the `BIO_gets()` or `BIO_puts()` functions.

The `BIO_flush()` function on a base64 BIO that is being written through is used to signal that no more data is to be encoded. This is used to flush the final block through the BIO.

The `BIO_FLAGS_BASE64_NO_NL` option can be set with `BIO_set_flags()` to encode the data all on one line or expect the data to be all on one line.

NOTES

Because of the format of base64 encoding the end of the encoded block cannot always be reliably determined.

RESTRICTIONS

The ambiguity of EOF in base64 encoded data can cause additional data following the base64 encoded block to be misinterpreted.

There should be some way of specifying a test that the BIO can perform to reliably determine EOF (for example, a MIME boundary).

RETURN VALUES

The `BIO_f_base64()` function returns the base64 BIO method.

EXAMPLES

Base64 encode the string "Hello World\n" and write the result to standard output:

```
BIO *bio, *b64;
char message[] = "Hello World \n";

b64 = BIO_new(BIO_f_base64());
bio = BIO_new_fp(stdout, BIO_NOCLOSE);
bio = BIO_push(b64, bio);

BIO_write(bio, message, strlen(message));
```

```
 BIO_flush(bio);
 BIO_free_all(bio);
```

Read Base64 encoded data from standard input and write the decoded data to standard output:

```
 BIO *bio, *b64, bio_out;
 char inbuf[512];
 int inlen;
 char message[] = "Hello World \n";

 b64 = BIO_new(BIO_f_base64());
 bio = BIO_new_fp(stdin, BIO_NOCLOSE);
 bio_out = BIO_new_fp(stdout, BIO_NOCLOSE);
 bio = BIO_push(b64, bio);

 while((inlen = BIO_read(bio, inbuf, strlen(message))) > 0)

BIO_write(bio_out, inbuf, inlen);

 BIO_free_all(bio);
```

BIO_f_buffer

NAME

BIO_f_buffer – Buffering BIO

SYNOPSIS

```
#include <openssl/bio.h>

BIO_METHOD * BIO_f_buffer(
       void
);

#define BIO_get_buffer_num_lines(b) BIO_ctrl(
       b,BIO_C_GET_BUFF_NUM_LINES,0,NULL
);

#define BIO_set_read_buffer_size(b,size) BIO_int_ctrl(
       b,BIO_C_SET_BUFF_SIZE,size,0
);

#define BIO_set_write_buffer_size(b,size) BIO_int_ctrl(
       b,BIO_C_SET_BUFF_SIZE,size,1
);

#define BIO_set_buffer_size(b,size)BIO_ctrl(
       b,BIO_C_SET_BUFF_SIZE,size,NULL
);

#define BIO_set_buffer_read_data(b,buf,num) BIO_ctrl(
       b,BIO_C_SET_BUFF_READ_DATA,num,buf
);
```

DESCRIPTION

The `BIO_f_buffer()` function returns the buffering BIO method.

Data written to a buffering BIO is buffered and periodically written to the next BIO in the chain. Data read from a buffering BIO comes from an internal buffer which is filled from the next BIO in the chain. Both `BIO_gets()` and `BIO_puts()` are supported.

Calling `BIO_reset()` on a buffering BIO clears any buffered data.

The `BIO_get_buffer_num_lines()` function returns the number of lines buffered.

The `BIO_set_read_buffer_size()`, `BIO_set_write_buffer_size()` and `BIO_set_buffer_size()` functions set the read, write or both read and write buffer sizes to `size`. The initial buffer size is `DEFAULT_BUFFER_SIZE`, currently 1024. Any attempt to reduce the buffer size below `DEFAULT_BUFFER_SIZE` is ignored. Any buffered data is cleared when the buffer is resized.

The `BIO_set_buffer_read_data()` function clears the read buffer and fills it with `num` bytes of `buf`. If `num` is larger than the current buffer size the buffer is expanded.

NOTES

Buffering BIOs implement `BIO_gets()` by using `BIO_read()` operations on the next BIO in the chain. By prepending a buffering BIO to a chain it is therefore possible to provide `BIO_gets()` functionality if the following BIOs do not support it (for example, SSL BIOs).

Data is only written to the next BIO in the chain when the write buffer fills or when `BIO_flush()` is called. It is therefore important to call `BIO_flush()` whenever any pending data should be written, such as when removing a buffering BIO using `BIO_pop()`. The `BIO_flush()` function might need to be retried if the ultimate source/sink BIO is non blocking.

RETURN VALUES

`BIO_f_buffer()` returns the buffering BIO method.

`BIO_get_buffer_num_lines()` returns the number of lines buffered (may be 0).

The `BIO_set_read_buffer_size()`, `BIO_set_write_buffer_size()`, and `BIO_set_buffer_size()` functions return 1 if the buffer was successfully resized, or 0 for failure.

The `BIO_set_buffer_read_data()` function returns 1 if the data was set correctly or 0 if there was an error.

BIO_f_cipher

NAME

BIO_f_cipher, BIO_set_cipher, BIO_get_cipher_status, BIO_get_cipher_ctx – Cipher BIO filter

SYNOPSIS

```
#include <openssl/bio.h>
#include <openssl/evp.h>

BIO_METHOD *BIO_f_cipher(
        void
);

void BIO_set_cipher(
        BIO *b,const EVP_CIPHER *cipher, unsigned char *key, unsigned char *iv, int enc
);

int BIO_get_cipher_status(
        BIO *b
);

int BIO_get_cipher_ctx(
        BIO *b, EVP_CIPHER_CTX **pctx
);
```

DESCRIPTION

The `BIO_f_cipher()` function returns the cipher BIO method. This is a filter BIO that encrypts any data written through it, and decrypts any data read from it. It is a BIO wrapper for the `EVP_CipherInit()`, `EVP_CipherUpdate()`, and `EVP_CipherFinal()` cipher functions .

Cipher BIOs do not support `BIO_gets()` or `BIO_puts()`.

The `BIO_flush()` function on an encryption BIO that is being written through is used to signal that no more data is to be encrypted. This is used to flush and possibly pad the final block through the BIO.

The `BIO_set_cipher()` function sets the cipher of BIO `b` to `cipher` using `key` and `iv`. The `enc` should be set to 1 for encryption and zero for decryption.

When reading from an encryption BIO the final block is automatically decrypted and checked when EOF is detected. The `BIO_get_cipher_status()` function is a `BIO_ctrl()` macro which can be called to determine whether the decryption operation was successful.

The `BIO_get_cipher_ctx()` function is a `BIO_ctrl()` macro which retrieves the internal BIO cipher context. The retrieved context can be used in conjunction with the standard cipher routines to set it up. This is useful when `BIO_set_cipher()` is not flexible enough for the applications needs.

NOTES

When encrypting, `BIO_flush()` must be called to flush the final block through the BIO. Otherwise, the final block will fail a subsequent decrypt.

When decrypting, an error on the final block is signalled by a zero return value from the read operation. A successful decrypt followed by EOF will also return zero for the final read. The `BIO_get_cipher_status()` function should be called to determine if the decrypt was successful.

If `BIO_gets()` or `BIO_puts()` support is needed then it can be achieved by preceding the cipher BIO with a buffering BIO.

RETURN VALUES

`BIO_f_cipher()` returns the cipher BIO method.

`BIO_set_cipher()` does not return a value.

`BIO_get_cipher_status()` returns 1 for a successful decrypt and 0 for failure.

`BIO_get_cipher_ctx()` always returns 1.

BIO_f_md

NAME

BIO_f_md, BIO_set_md , BIO_get_md, BIO_get_md_ctx – Message digest BIO filter

SYNOPSIS

```
#include <openssl/bio.h>
#include <openssl/evp.h>

BIO_METHOD *BIO_f_md(
        void
);

int BIO_set_md(
        BIO *b,EVP_MD *md
);

int BIO_get_md(
        BIO *b,EVP_MD **mdp
);

int BIO_get_md(
        BIO *b,EVP_MD_CTX **mdcp
);
```

DESCRIPTION

BIO_f_md() returns the message digest BIO method. This is a filter BIO that digests any data passed through it, it is a BIO wrapper for the EVP_DigestInit(), EVP_DigestUpdate(), and EVP_DigestFinal() digest functions.

Any data written or read through a digest BIO using BIO_read() and BIO_write() is digested.

The BIO_gets() function, if its size parameter is large enough, finishes the digest calculation and returns the digest value. The BIO_puts() function is not supported.

The BIO_reset() function reinitializes a digest BIO.

The BIO_set_md() function sets the message digest of BIO b to md. This must be called to initialize a digest BIO before any data is passed through it. It is a BIO_ctrl() macro.

The BIO_get_md() function places the a pointer to the digest BIOs digest method in mdp. It is a BIO_ctrl() macro.

The BIO_get_md_ctx() function returns the digest BIOs context into mdcp.

NOTES

The context returned by the BIO_get_md_ctx() function can be used in calls to the EVP_DigestFinal(), EVP_SignFinal(), and EVP_VerifyFinal() functions.

The context returned by the BIO_get_md_ctx() function is an internal context structure. Changes made to this context will affect the digest BIO itself and the context pointer will become invalid when the digest BIO is freed.

After the digest has been retrieved from a digest BIO it must be reinitialized by calling `BIO_reset()` or `BIO_set_md()` before any more data is passed through it.

If an application needs to call `BIO_gets()` or `BIO_puts()` through a chain containing digest BIOs then this can be done by prepending a buffering BIO.

RESTRICTIONS

The lack of support for `BIO_puts()` and the nonstandard behavior of `BIO_gets()` could be regarded as anomalous. It could be argued that `BIO_gets()` and `BIO_puts()` should be passed to the next BIO in the chain and digest the data passed through and that digests should be retrieved using a separate `BIO_ctrl()` call.

RETURN VALUES

The `BIO_f_md()` function returns the digest BIO method.

The `BIO_set_md()`, `BIO_get_md()`, and `BIO_md_ctx()` functions return 1 for success and 0 for failure.

EXAMPLES

Create a BIO chain containing an SHA1 and MD5 digest BIO and pass the string "Hello World" through it. (Error checking has been omitted for clarity.)

```
BIO *bio, *mdtmp;
char message[] = "Hello World";
bio = BIO_new(BIO_s_null());
mdtmp = BIO_new(BIO_f_md());
BIO_set_md(mdtmp, EVP_sha1());

/* For BIO_push() we want to append the sink BIO and keep a note of
 * the start of the chain.
 */

bio = BIO_push(mdtmp, bio);
mdtmp = BIO_new(BIO_f_md());
BIO_set_md(mdtmp, EVP_md5());
bio = BIO_push(mdtmp, bio);

/* Note: mdtmp can now be discarded */

BIO_write(bio, message, strlen(message));
```

Digest data by reading through a chain:

```
BIO *bio, *mdtmp;
char buf[1024];
int rdlen;
bio = BIO_new_file(file, "rb");
mdtmp = BIO_new(BIO_f_md());
BIO_set_md(mdtmp, EVP_sha1());
bio = BIO_push(mdtmp, bio);
mdtmp = BIO_new(BIO_f_md());
BIO_set_md(mdtmp, EVP_md5());
bio = BIO_push(mdtmp, bio);
do {

rdlen = BIO_read(bio, buf, sizeof(buf));
```

```
        /* Might want to do something with the data here */

 } while(rdlen > 0);
```

Retrieve the message digests from a BIO chain and output them. This could be used with the previous examples:

```
 BIO *mdtmp;
 unsigned char mdbuf[EVP_MAX_MD_SIZE];
 int mdlen;
 int i;
 mdtmp = bio;/* Assume bio has previously been set up */
 do {

EVP_MD *md;

 mdtmp = BIO_find_type(mdtmp, BIO_TYPE_MD);
        if(!mdtmp) break;

BIO_get_md(mdtmp, &md);
        printf("%s digest", OBJ_nid2sn(EVP_MD_type(md)));

mdlen = BIO_gets(mdtmp, mdbuf, EVP_MAX_MD_SIZE);
for(i = 0; i < mdlen; i++) printf(":%02X", mdbuf[i]);
printf("\n");

mdtmp = BIO_next(mdtmp);
 } while(mdtmp);

 BIO_free_all(bio);
```

BIO_f_null

NAME

BIO_f_null – Null filter

SYNOPSIS

```
#include <openssl/bio.h>
BIO_METHOD *BIO_f_null(
       void
);
```

DESCRIPTION

`BIO_f_null()` returns the null filter BIO method. This is a filter BIO that does nothing.

All requests to a null filter BIO are passed through to the next BIO in the chain. This means that a BIO chain containing a null filter BIO behaves just as though the BIO was not there.

NOTES

A null filter BIO is not particularly useful.

RETURN VALUES

The `BIO_f_null()` function returns the null filter BIO method.

BIO_f_ssl

NAME

BIO_f_ssl, BIO_set_ssl, BIO_get_ssl BIO_set_ssl_mode, BIO_set_ssl_renegotiate_bytes, BIO_get_num_renegotiates, BIO_set_ssl_renegotiate_timeout, BIO_new_ssl, BIO_new_ssl_connect, BIO_new_buffer_ssl_connect, BIO_ssl_copy_session_id, BIO_ssl_shutdown – SSL BIO

SYNOPSIS

```
#include <openssl/bio.h>
#include <openssl/ssl.h>

BIO_METHOD *BIO_f_ssl(
        void
);

#define BIO_set_ssl(b,ssl,c)BIO_ctrl(b,BIO_C_SET_SSL,c,(char *)ssl)
#define BIO_get_ssl(b,sslp)BIO_ctrl(b,BIO_C_GET_SSL,0,(char *)sslp)
#define BIO_set_ssl_mode(b,client)BIO_ctrl(b,BIO_C_SSL_MODE,client,NULL)
#define BIO_set_ssl_renegotiate_bytes(b,num) \

BIO_ctrl(
        b,BIO_C_SET_SSL_RENEGOTIATE_BYTES,num,NULL
);

#define BIO_set_ssl_renegotiate_timeout(b,seconds) \

BIO_ctrl(
        b,BIO_C_SET_SSL_RENEGOTIATE_TIMEOUT,seconds,NULL
);

#define BIO_get_num_renegotiates(b) \

BIO_ctrl(
        b,BIO_C_SET_SSL_NUM_RENEGOTIATES,0,NULL
);

BIO *BIO_new_ssl(
        SSL_CTX *ctx,int client
);

BIO *BIO_new_ssl_connect(
        SSL_CTX *ctx
);

BIO *BIO_new_buffer_ssl_connect(
        SSL_CTX *ctx
);

int BIO_ssl_copy_session_id(
        BIO *to,BIO *from
);

void BIO_ssl_shutdown(
```

```
            BIO *bio
      );
#define BIO_do_handshake(b)BIO_ctrl(b,BIO_C_DO_STATE_MACHINE,0,NULL)
```

DESCRIPTION

The BIO_f_ssl() function returns the SSL BIO method. This is a filter BIO which is a wrapper round the OpenSSL SSL routines adding a BIO "flavor" to SSL I/O.

I/O performed on an SSL BIO communicates using the SSL protocol with the SSL's read and write BIOs. If an SSL connection is not established, then an attempt is made to establish one on the first I/O call.

If a BIO is appended to an SSL BIO using the BIO_push() function, it is automatically used as the SSL BIO's read and write BIOs.

Calling BIO_reset() on an SSL BIO closes down any current SSL connection by calling SSL_shutdown(). BIO_reset() is then sent to the next BIO in the chain. This typically will disconnect the underlying transport. The SSL BIO is then reset to the initial accept or connect state.

If the close flag is set when an SSL BIO is freed then the internal SSL structure is also freed using SSL_free().

The BIO_set_ssl() function sets the internal SSL pointer of BIO b to ssl using the close (c) option .

The BIO_get_ssl() function retrieves the SSL pointer of BIO b. It then can be manipulated using the standard SSL library functions.

The BIO_set_ssl_mode() function sets the SSL BIO mode to client. If client is 1, client mode is set. If client is 0, server mode is set.

The BIO_set_ssl_renegotiate_bytes() function sets the renegotiate byte count to num. When set after every num bytes of I/O (read and write) the SSL session is automatically renegotiated. The num value must be at least 512 bytes.

The BIO_set_ssl_renegotiate_timeout() function sets the renegotiate timeout to seconds. When the renegotiate timeout elapses the session is automatically renegotiated.

The BIO_get_num_renegotiates() function returns the total number of session renegotiations due to I/O or timeout.

The BIO_new_ssl() function allocates an SSL BIO using SSL_CTX ctx and using client mode if client is not zero.

The BIO_new_ssl_connect() funciton creates a new BIO chain consisting of an SSL BIO (using ctx) followed by a connect BIO.

The BIO_new_buffer_ssl_connect() function creates a new BIO chain consisting of a buffering BIO, an SSL BIO (using ctx), and a connect BIO.

The BIO_ssl_copy_session_id() function copies an SSL session id between BIO chains from and to. It does this by locating the SSL BIOs in each chain and calling SSL_copy_session_id() on the internal SSL pointer.

The BIO_ssl_shutdown() function closes down an SSL connection on BIO chain bio. It does this by locating the SSL BIO in the chain and calling SSL_shutdown() on its internal SSL pointer.

The BIO_do_handshake() function attempts to complete an SSL handshake on the supplied BIO and establish the SSL connection. It returns 1 if the connection was established successfully. A zero or negative value is returned if the connection could not be established, the BIO_should_retry() function should be used for non blocking connect BIOs to determine if the call should be retried. If an SSL connection has already been established this call has no effect.

NOTES

SSL BIOs are exceptional in that if the underlying transport is non blocking they can still request a retry in exceptional circumstances. Specifically this will happen if a session renegotiation takes place during a `BIO_read()` operation, one case where this happens is when SGC or step up occurs.

In OpenSSL 0.9.6 and later the `SSL_AUTO_RETRY` option can be set to disable this behavior. That is, when this flag is set, an SSL BIO using a blocking transport will never request a retry.

Since unknown `BIO_ctrl()` operations are sent through filter BIOs the servers name and port can be set using the `BIO_set_host()` function on the BIO returned by the `BIO_new_ssl_connect()` function without having to locate the connect BIO first.

Applications do not have to call `BIO_do_handshake()`, but may wish to do so to separate the handshake process from other I/O processing.

RETURN VALUES

EXAMPLE

This SSL/TLS client example attempts to retrieve a page from an SSL/TLS web server. The I/O routines are identical to those of the unencrypted example in *BIO_s_connect*.

```
BIO *sbio, *out;
int len;
char tmpbuf[1024];
SSL_CTX *ctx;
SSL *ssl;

ERR_load_crypto_strings();
ERR_load_SSL_strings();
OpenSSL_add_all_algorithms();

/* We would seed the PRNG here if the platform didn't
 * do it automatically
 */

ctx = SSL_CTX_new(SSLv23_client_method());

/* We'd normally set some stuff like the verify paths and
 * mode here because as things stand this will connect to
 * any server whose certificate is signed by any CA.
 */

sbio = BIO_new_ssl_connect(ctx);

BIO_get_ssl(sbio, &amp;ssl);

if(!ssl) {
  fprintf(stderr, "Can't locate SSL pointer\n");
  /* whatever ... */
}

/* Don't want any retries */

SSL_set_mode(ssl, SSL_MODE_AUTO_RETRY);
```

```
 /* We might want to do other things with ssl here */

 BIO_set_conn_hostname(sbio, "localhost:https");
 out = BIO_new_fp(stdout, BIO_NOCLOSE);
 if(BIO_do_connect(sbio) <= 0) {

fprintf(stderr, "Error connecting to server\n");
ERR_print_errors_fp(stderr);

/* whatever ... */

 }

 if(BIO_do_handshake(sbio) <= 0) {
fprintf(stderr, "Error establishing SSL connection\n");
ERR_print_errors_fp(stderr);

/* whatever ... */

 }

 /* Could examine ssl here to get connection info */

 BIO_puts(sbio, "GET / HTTP/1.0\n\n");
 for(;;) {

len = BIO_read(sbio, tmpbuf, 1024);
if(len <= 0) break;
BIO_write(out, tmpbuf, len);
 }

 BIO_free_all(sbio);
 BIO_free(out);
```

This server example makes use of a buffering BIO to allow lines to be read from the SSL BIO using `BIO_gets`. It creates a pseudo web page containing the actual request from a client and also echoes the request to standard output.

```
 BIO *sbio, *bbio, *acpt, *out;
 int len;
 char tmpbuf[1024];
 SSL_CTX *ctx;
 SSL *ssl;

 ERR_load_crypto_strings();
 ERR_load_SSL_strings();
 OpenSSL_add_all_algorithms();

 /* Might seed PRNG here */

 ctx = SSL_CTX_new(SSLv23_server_method());

 if (!SSL_CTX_use_certificate_file(ctx,"server.pem",SSL_FILETYPE_PEM)

|| !SSL_CTX_use_PrivateKey_file(ctx,"server.pem",SSL_FILETYPE_PEM)
|| !SSL_CTX_check_private_key(ctx)) {
```

```
fprintf(stderr, "Error setting up SSL_CTX\n");
ERR_print_errors_fp(stderr);
return 0;
 }

 /* Might do other things here like setting verify locations and
  * DH and/or RSA temporary key callbacks
  */

 /* New SSL BIO setup as server */

 sbio=BIO_new_ssl(ctx,0);
 BIO_get_ssl(sbio, &amp;ssl);

 if(!ssl) {
   fprintf(stderr, "Can't locate SSL pointer\n");

   /* whatever ... */

 }

 /* Don't want any retries */

 SSL_set_mode(ssl, SSL_MODE_AUTO_RETRY);

 /* Create the buffering BIO */

 bbio = BIO_new(BIO_f_buffer());

 /* Add to chain */

 sbio = BIO_push(bbio, sbio);
 acpt=BIO_new_accept("4433");

 /* By doing this when a new connection is established
  * we automatically have sbio inserted into it. The
  * BIO chain is now 'swallowed' by the accept BIO and
  * will be freed when the accept BIO is freed.
  */

 BIO_set_accept_bios(acpt,sbio);
 out = BIO_new_fp(stdout, BIO_NOCLOSE);

 /* Setup accept BIO */

 if(BIO_do_accept(acpt) <= 0) {

fprintf(stderr, "Error setting up accept BIO\n");

ERR_print_errors_fp(stderr);
return 0;

}

 /* Now wait for incoming connection */

 if(BIO_do_accept(acpt) <= 0) {
```

```
fprintf(stderr, "Error in connection\n");

ERR_print_errors_fp(stderr);
return 0;

 }

 /* We only want one connection so remove and free
  * accept BIO
  */

 sbio = BIO_pop(acpt);

 BIO_free_all(acpt);

 if(BIO_do_handshake(sbio) <= 0) {

fprintf(stderr, "Error in SSL handshake\n");
ERR_print_errors_fp(stderr);
return 0;

 }

 BIO_puts(sbio, "HTTP/1.0 200 OK\r\nContent-type: text/html\r\n\r\n");
 BIO_puts(sbio, "&lt;pre>\r\nConnection Established\r\nRequest headers:\r\n");

 BIO_puts(sbio, "--------------------------------------------------\r\n");
 for(;;) {

 len = BIO_gets(sbio, tmpbuf, 1024);
        if(len <= 0) break;

BIO_write(sbio, tmpbuf, len);
BIO_write(out, tmpbuf, len);

/* Look for blank line signifying end of headers*/

if((tmpbuf[0] == '\r') || (tmpbuf[0] == '\n')) break;

 }

 BIO_puts(sbio, "--------------------------------------------------\r\n");
 BIO_puts(sbio, "&lt;/pre>\r\n");

 /* Since there is a buffering BIO present we had better flush it */

 BIO_flush(sbio);
 BIO_free_all(sbio);
```

BIO_find_type

NAME

BIO_find_type, BIO_next – BIO chain traversal

SYNOPSIS

```
#include <openssl/bio.h>

BIO *BIO_find_type(
        BIO *b,int bio_type)

BIO *BIO_next(
        BIO *b
);
```

```
#define BIO_method_type(b)((b)->method->type)
#define BIO_TYPE_NONE0
#define BIO_TYPE_MEM(1|0x0400)
#define BIO_TYPE_FILE(2|0x0400

#define BIO_TYPE_FD(4|0x0400|0x0100)
#define BIO_TYPE_SOCKET(5|0x0400|0x0100)
#define BIO_TYPE_NULL(6|0x0400)
#define BIO_TYPE_SSL(7|0x0200)
#define BIO_TYPE_MD(8|0x0200)
#define BIO_TYPE_BUFFER(9|0x0200)
#define BIO_TYPE_CIPHER(10|0x0200)
#define BIO_TYPE_BASE64(11|0x0200)
#define BIO_TYPE_CONNECT(12|0x0400|0x0100)
#define BIO_TYPE_ACCEPT(13|0x0400|0x0100)
#define BIO_TYPE_PROXY_CLIENT(14|0x0200)
#define BIO_TYPE_PROXY_SERVER(15|0x0200)
#define BIO_TYPE_NBIO_TEST(16|0x0200)
#define BIO_TYPE_NULL_FILTER(17|0x0200)
#define BIO_TYPE_BER(18|0x0200)
#define BIO_TYPE_BIO(19|0x0400)

#define BIO_TYPE_DESCRIPTOR0x0100
#define BIO_TYPE_FILTER0x0200
#define BIO_TYPE_SOURCE_SINK0x0400
```

DESCRIPTION

The `BIO_find_type()` function searches for a BIO of a given type in a chain, starting at BIO `b`. If `type` is a specific type, such as `BIO_TYPE_MEM`, then a search is made for a BIO of that type. If `type` is a general type, such as `BIO_TYPE_SOURCE_SINK`, then the next matching BIO of the given general type is sought. The `BIO_find_type()`function returns the next matching BIO or NULL if none is found.

NOTE Some `BIO_TYPE_*` types do not have corresponding BIO implementations.

The `BIO_next()`function returns the next BIO in a chain. It can be used to traverse all BIOs in a chain or used in conjunction with the `BIO_find_type()` function to find all BIOs of a certain type.

The `BIO_method_type()` function returns the type of a BIO.

NOTES

The `BIO_next()` function was added to OpenSSL 0.9.6 to provide a clean way to traverse a BIO chain or find multiple matches using the `BIO_find_type()` function. Previous versions used the following:

```
next = bio->next_bio;
```

RESTRICTIONS

The `BIO_find_type()` function in OpenSSL 0.9.5a and earlier could not be safely passed a NULL pointer for the `b` argument.

RETURN VALUES

The `BIO_find_type()` function returns a matching BIO or NULL for no match.

The `BIO_next()` function returns the next BIO in a chain.

The `BIO_method_type()` function returns the type of the BIO `b`.

EXAMPLE

Traverse a chain looking for digest BIOs:

```
 BIO *btmp;
 btmp = in_bio;/* in_bio is chain to search through */

 do {
 btmp = BIO_find_type(btmp, BIO_TYPE_MD);
if(btmp == NULL) break;/* Not found */

/* btmp is a digest BIO, do something with it ...*/

   ...

btmp = BIO_next(btmp);
 } while(btmp);
```

BIO_new

NAME

BIO_new, BIO_set, BIO_free, BIO_vfree, BIO_free_all – BIO allocation and freeing functions

SYNOPSIS

```
#include <openssl/bio.h>

BIO * BIO_new(
        BIO_METHOD *type
);

intBIO_set(
        BIO *a,BIO_METHOD *type
);

intBIO_free(
        BIO *a
);

voidBIO_vfree(
        BIO *a
);

voidBIO_free_all(
        BIO *a
);
```

DESCRIPTION

The `BIO_new()` function returns a new BIO using method `type`.

The `BIO_set()` function sets the method of an already existing BIO.

The `BIO_free()` function frees up a single BIO. The `BIO_vfree()`function also frees up a single BIO but it does not return a value. Calling `BIO_free()` functon might also have some effect on the underlying I/O structure, for example it may close the file being referred to under certain circumstances. For more details see the individual BIO_METHOD descriptions.

The `BIO_free_all()` function frees up an entire BIO chain. It does not halt if an error occurs freeing up an individual BIO in the chain.

NOTES

Some BIOs, such as memory BIOs, can be used immediately after calling `BIO_new()`. Others, such as file BIOs, need some additional initialization, and frequently a utility function exists to create and initialize such BIOs.

If the `BIO_free()` function is called on a BIO chain it will only free one BIO, resulting in a memory leak.

Calling `BIO_free_all()` a single BIO has the same effect as calling `BIO_free()` on it other than the discarded return value.

Usually the `type` argument is supplied by a function which returns a pointer to a `BIO_METHOD`. There is a naming convention for such functions: a source/sink BIO is usually called `BIO_s_*()`, and a filter is called `BIO BIO_f_*()`.

RETURN VALUES

The `BIO_new()` function returns a newly created BIO or NULL if the call fails.

The `BIO_set()` and `BIO_free()` functions return 1 for success and 0 for failure.

The `BIO_free_all()` and `BIO_vfree()` functions do not return values.

EXAMPLE

Create a memory BIO:

```
BIO *mem = BIO_new(BIO_s_mem());
```

BIO_new_bio_pair

NAME

BIO_new_bio_pair – Create a new BIO pair

SYNOPSIS

```
#include <openssl/bio.h>
int BIO_new_bio_pair(
        BIO **bio1, size_t writebuf1, BIO **bio2, size_t writebuf2
);
```

DESCRIPTION

The `BIO_new_bio_pair()` function creates a buffering BIO pair. It has two endpoints between data can be buffered. Its typical use is to connect one endpoint as underlying input/output BIO to an SSL and access the other one controlled by the program instead of accessing the network connection directly.

The two new BIOs, `bio1` and `bio2`, are symmetric with respect to their functionality. The size of their buffers is determined by `writebuf1` and `writebuf2`. If the size given is 0, the default size is used.

The `BIO_new_bio_pair()` function does not check whether `bio1` or `bio2` point to another BIO. The values are overwritten, and the `BIO_free()` function is not called.

The two BIOs, even though forming a BIO pair, must be freed separately, using the `BIO_free()` function. This is important because some SSL functions, such as `SSL_set_bio()` and `SSL_free()`, call `BIO_free()` implicitly, so that the peer-BIO is left untouched and also must be freed using `BIO_free()`.

NOTES

As the data is buffered, the `SSL_operation()` function might return an `ERROR_SSL_WANT_READ` condition, but there is still data in the write buffer. An application must not rely on the error value of the `SSL_operation()` function, but must assure that the write buffer is always flushed first. Otherwise, a deadlock may occur as the peer might be waiting for the data before being able to continue.

RETURN VALUES

The following return values can occur:

- 1

  The BIO pair was created successfully. The new BIOs are available in `bio1` and `bio2`.

- 0

  The operation failed. The NULL pointer is stored into the locations for `bio1` and `bio2`. Check the error stack for more information.

EXAMPLE

The BIO pair can be used to have full control over the network access of an application. The application can call `select()` on the socket as required without having to go through the SSL interface.

```
BIO *internal_bio, *network_bio;
...

BIO_new_bio_pair(internal_bio, 0, network_bio, 0);
SSL_set_bio(ssl, internal_bio);
SSL_operations();
...

application |   TLS-engine
   |        |
   +----------> SSL_operations()
            |     /\    ||
            |     ||    \/
            |   BIO-pair (internal_bio)
   +----------< BIO-pair (network_bio)
   |        |
 socket     |

 ...

 SSL_free(ssl);/* implicitly frees internal_bio */
 BIO_free(network_bio);
 ...
```

As the BIO pair will only buffer the data and never directly access the connection, it behaves non-blocking and will return as soon as the write buffer is full or the read buffer is drained. Then the application has to flush the write buffer and/or fill the read buffer.

Use the `BIO_ctrl_pending()` function to find out whether data is buffered in the BIO and must be transfered to the network. Use the `BIO_ctrl_get_read_request()` function to find out how many bytes must be written into the buffer before the `SSL_operation()` can continue.

SEE ALSO

Functions: *SSL_set_bio*, *ssl*, *bio*, *BIO_ctrl_pending*, *BIO_ctrl_get_read_request*

BIO_push

NAME

BIO_push, BIO_pop – Add and remove BIOs from a chain.

SYNOPSIS

```
#include <openssl/bio.h>
BIO * BIO_push(
        BIO *b,BIO *append
);
BIO * BIO_pop(
        BIO *b
);
```

DESCRIPTION

The `BIO_push()` function appends the BIO `append` to `b`. It returns `b`.

The `BIO_pop()` function removes the BIO `b` from a chain and returns the next BIO in the chain. The return is NULL if there are no more BIOs in the chain. The removed BIO then becomes a single BIO with no association to the original chain. It can be freed or attached to a different chain.

NOTES

The names of these functions are somewhat misleading. The `BIO_push()` function joins two BIO chains, whereas the `BIO_pop()` function deletes a single BIO from a chain. The deleted BIO does not need to be at the end of a chain.

The process of calling the `BIO_push()` and `BIO_pop()` functions on a BIO may have additional consequences (a control call is made to the affected BIOs). Any effects are noted in the descriptions of individual BIOs.

RETURN VALUES

The `BIO_push()` function returns the end of the chain, `b`.

The `BIO_pop()` function returns the next BIO in the chain. If there are no more BIOs in the chain, the return is NULL.

EXAMPLES

For these examples, `md1` and `md2` are digest BIOs, `b64` is a base64 BIO, and `f` is a file BIO.

If the call:

```
BIO_push(b64, f);
```

is made then the new chain will be `b64-chain`. After making the following calls

```
BIO_push(md2, b64);
BIO_push(md1, md2);
```

the new chain is `md1-md2-b64-f`. Data written to `md1` will be digested by `md1` and `md2`, `base64` encoded and written to `f`.

NOTE Reading causes data to pass in the reverse direction. Data is read from `f`, `base64` decoded and digested by `md1` and `md2`.

If the call:

```
BIO_pop(md2);
```

is made, the call will return `b64`, and the new chain will be `md1-b64-f`. Data can be written to `md1` as before.

BIO_read

NAME

BIO_read, BIO_write, BIO_gets , BIO_puts – BIO I/O functions

SYNOPSIS

```
#include <openssl/bio.h>
 intBIO_read(
       BIO *b, void *buf, int len
);
 intBIO_gets(
       BIO *b,char *buf, int size
);
 intBIO_write(
       BIO *b, const void *buf, int len
);
 intBIO_puts(
       BIO *b,const char *buf
);
```

DESCRIPTION

The `BIO_read()` function attempts to read `len` bytes from BIO `b` and places the data in `buf`.

The `BIO_gets()` function performs the BIOs gets operation and places the data in `buf`. Usually this operation will attempt to read a line of data from the BIO of maximum length `len`. However, there are exceptions to this. For example, `BIO_gets()` on a digest BIO will calculate and return the digest, and other BIOs might not support `BIO_gets()`.

The `BIO_write()` function attempts to write `len` bytes from `buf` to BIO `b`.

The `BIO_puts()` function attempts to write a null terminated string `buf` to BIO `b`

NOTES

A 0 or -1 return might indicate an error. However, when the source/sink is non-blocking or of a certain type, it might be an indication that no data is available and that the application should retry the operation later.

One technique sometimes used with blocking sockets is to use a system call (such as `select()`, `poll()`, or equivalent) to determine when data is available, and then call `read()` to read the data. The equivalent with BIOs (that is, call `select()` on the underlying I/O structure and then call `BIO_read()` to read the data) should not be used because a single call to `BIO_read()` can cause several reads (and writes in the case of SSL BIOs) on the underlying I/O structure and may block as a result. Instead `select()` (or equivalent) should be combined with nonblocking I/O so successive reads will request a retry instead of blocking.

See *BIO_should_retry* for details of how to determine the cause of a retry and other I/O issues.

If the `BIO_gets()` function is not supported by a BIO then it is possible to work around this by adding a buffering BIO, `BIO_f_buffer()`, to the chain.

RETURN VALUES

All these functions return either the amount of data successfully read or written (if the return value is positive) or that no data was successfully read or written if the result is 0 or -1. If the return value is -2 then the operation is not implemented in the specific BIO type.

SEE ALSO

Functions: *BIO_should_retry*

BIO_s_accept

NAME

BIO_s_accept, BIO_set_nbio, BIO_set_accept_port, BIO_get_accept_port, BIO_set_nbio_accept, BIO_set_accept_bios, BIO_set_bind_mode, BIO_get_bind_mode, BIO_do_accept – Accept BIO

SYNOPSIS

```
#include <openssl/bio.h>

BIO_METHOD * BIO_s_accept(
       void
);

#define BIO_set_accept_port(b,name) BIO_ctrl(b,BIO_C_SET_ACCEPT,0,(char *)name)
#define BIO_get_accept_port(b)BIO_ptr_ctrl(b,BIO_C_GET_ACCEPT,0)

BIO *BIO_new_accept(
       char *host_port
);

#define BIO_set_nbio_accept(b,n) BIO_ctrl(b,BIO_C_SET_ACCEPT,1,(n)?"a":NULL)
#define BIO_set_accept_bios(b,bio) BIO_ctrl(b,BIO_C_SET_ACCEPT,2,(char *)bio)
#define BIO_set_bind_mode(b,mode) BIO_ctrl(b,BIO_C_SET_BIND_MODE,mode,NULL)
#define BIO_get_bind_mode(b,mode) BIO_ctrl(b,BIO_C_GET_BIND_MODE,0,NULL)
#define BIO_BIND_NORMAL0
#define BIO_BIND_REUSEADDR_IF_UNUSED1
#define BIO_BIND_REUSEADDR2
#define BIO_do_accept(b)BIO_do_handshake(b)
```

DESCRIPTION

The `BIO_s_accept()` function returns the accept BIO method. This is a wrapper round the platform's TCP/IP socket accept routines.

Using accept BIOs, TCP/IP connections can be accepted and data transferred using only BIO routines. In this way any platform specific operations are hidden by the BIO abstraction.

Read and write operations on an accept BIO will perform I/O on the underlying connection. If no connection is established and the port is set up properly then the BIO waits for an incoming connection.

Accept BIOs support `BIO_puts()` but not `BIO_gets()`.

If the close option is set on an accept BIO then any active connection on that chain is shutdown and the socket closed when the BIO is freed.

Calling `BIO_reset()` on a accept BIO will close any active connection and reset the BIO into a state where it awaits another incoming connection.

The `BIO_get_fd()` and `BIO_set_fd()` functions can be called to retrieve or set the accept socket. See *BIO_s_fd*

`BIO_set_accept_port()` uses the string `name` to set the accept port. The port is represented as a string of the form `host:port`, where host is the interface to use and port is the port. Either or both values can be `*` which is interpreted as meaning any interface or port respectively. Port has the same syntax as the port specified in `BIO_set_conn_port()` for connect BIOs; it can be a numerical port string or a string to lookup using `getservbyname()` and a string table.

BIO_new_accept() combines BIO_new() and BIO_set_accept_port() into a single call; it creates a new accept BIO with port host_port.

BIO_set_nbio_accept() sets the accept socket to blocking mode (the default) if n is 0 or nonblocking mode if n is 1.

The BIO_set_accept_bios() function can be used to set a chain of BIOs which will be duplicated and prepended to the chain when an incoming connection is received. This is useful if, for example, a buffering or SSL BIO is required for each connection. The chain of BIOs must not be freed after this call. They will be automatically freed when the accept BIO is freed.

The BIO_set_bind_mode() and BIO_get_bind_mode() functions set and retrieve the current bind mode. If BIO_BIND_NORMAL (the default) is set, then another socket cannot be bound to the same port. If BIO_BIND_REUSEADDR is set, then other sockets can bind to the same port. If BIO_BIND_REUSEADDR_IF_UNUSEDis set, then an attempt is first made to use BIO_BIN_NORMAL. If this fails and the port is not in use, then a second attempt is made using BIO_BIND_REUSEADDR.

BIO_do_accept() serves two functions. When it is first called, after the accept BIO has been setup, it will attempt to create the accept socket and bind an address to it. Second and subsequent calls to BIO_do_accept() will await an incoming connection.

NOTES

When an accept BIO is at the end of a chain it will await an incoming connection before processing I/O calls. When an accept BIO is not at then end of a chain it passes I/O calls to the next BIO in the chain.

When a connection is established a new socket BIO is created for the connection and appended to the chain. The chain is now accept->socket. This effectively means that attempting I/O on an initial accept socket will await an incoming connection then perform I/O on it.

If any additional BIOs have been set using the BIO_set_accept_bios() function then they are placed between the socket and the accept BIO. The chain will be accept->otherbios->socket.

If a server wishes to process multiple connections (as is normally the case), then the accept BIO must be made available for further incoming connections. This can be done by waiting for a connection and then calling:

```
connection = BIO_pop(accept);
```

After this call, connection will contain a BIO for the recently established connection and accept will be a single BIO again which can be used to await further incoming connections. If no further connections will be accepted, the accept can be freed using the BIO_free() function.

If only a single connection will be processed it is possible to perform I/O using the accept BIO. This is often undesirable however because the accept BIO will still accept additional incoming connections. This can be resolved by using the BIO_pop() function and freeing up the accept BIO after the initial connection.

RETURN VALUES

EXAMPLES

This example accepts two connections on port 4444, sends messages down each and finally closes both down.

```
BIO *abio, *cbio, *cbio2;
ERR_load_crypto_strings();
abio = BIO_new_accept("4444");
```

```
 /* First call to BIO_accept() sets up accept BIO */

 if(BIO_do_accept(abio) <= 0) {
fprintf(stderr, "Error setting up accept\n");
ERR_print_errors_fp(stderr);
exit;
 }

 /* Wait for incoming connection */
 if(BIO_do_accept(abio) <= 0) {
fprintf(stderr, "Error accepting connection\n");
ERR_print_errors_fp(stderr);
exit;
 }

 fprintf(stderr, "Connection 1 established\n");

 /* Retrieve BIO for connection */
 cbio = BIO_pop(abio);
 BIO_puts(cbio, "Connection 1: Sending out Data on initial connection\n");
 fprintf(stderr, "Sent out data on connection 1\n");

 /* Wait for another connection */
 if(BIO_do_accept(abio) <= 0) {
fprintf(stderr, "Error accepting connection\n");
ERR_print_errors_fp(stderr);
exit;
 }

 fprintf(stderr, "Connection 2 established\n");

 /* Close accept BIO to refuse further connections */

 cbio2 = BIO_pop(abio);
 BIO_free(abio);
 BIO_puts(cbio2, "Connection 2: Sending out Data on second\n");
 fprintf(stderr, "Sent out data on connection 2\n");

 BIO_puts(cbio, "Connection 1: Second connection established\n");

 /* Close the two established connections */

 BIO_free(cbio);
 BIO_free(cbio2);
```

BIO_s_bio

NAME

BIO_s_bio, BIO_make_bio_pair, BIO_destroy_bio_pair, BIO_shutdown_wr, BIO_set_write_buf_size BIO_get_write_buf_size, BIO_new_bio_pair, BIO_get_write_guarantee, BIO_ctrl_get_write_guarantee, BIO_get_read_request, BIO_ctrl_get_read_request, BIO_ctrl_reset_read_request – BIO pair BIO

SYNOPSIS

```
#include <openssl/bio.h>

BIO_METHOD *BIO_s_bio(
        void
);

#define BIO_make_bio_pair(b1,b2)   (int)BIO_ctrl(b1,BIO_C_MAKE_BIO_PAIR,0,b2)
#define BIO_destroy_bio_pair(b)    (int)BIO_ctrl(b,BIO_C_DESTROY_BIO_PAIR,0,NULL)
#define BIO_shutdown_wr(b) (int)BIO_ctrl(b, BIO_C_SHUTDOWN_WR, 0, NULL)
#define BIO_set_write_buf_size(b,size) (int)BIO_ctrl(b,BIO_C_SET_WRITE_BUF_SIZE,size,NULL)
#define BIO_get_write_buf_size(b,size) (size_t)BIO_ctrl(b,BIO_C_GET_WRITE_BUF_SIZE,size,NULL)

int BIO_new_bio_pair(
        BIO **bio1, size_t writebuf1, BIO **bio2, size_t writebuf2
);

#define BIO_get_write_guarantee(b) (int)BIO_ctrl(b,BIO_C_GET_WRITE_GUARANTEE,0,NULL)

size_t BIO_ctrl_get_write_guarantee(
        BIO *b
);

#define BIO_get_read_request(b)    (int)BIO_ctrl(b,BIO_C_GET_READ_REQUEST,0,NULL)

size_t BIO_ctrl_get_read_request(
        BIO *b
);

int BIO_ctrl_reset_read_request(
        BIO *b
);
```

DESCRIPTION

The `BIO_s_bio()` function returns the method for a BIO pair. A BIO pair is a pair of source/sink BIOs where data written to either half of the pair is buffered and can be read from the other half. Both halves must usually by handled by the same application thread since no locking is done on the internal data structures.

Since BIO chains typically end in a source/sink BIO it is possible to make this one half of a BIO pair and have all the data processed by the chain under application control.

One typical use of BIO pairs is to place TLS/SSL I/O under application control, this can be used when the application wishes to use a non standard transport for TLS/SSL or the normal socket routines are inappropriate.

Calls to `BIO_read()` will read data from the buffer or request a retry if no data is available.

Calls to BIO_write() will place data in the buffer or request a retry if the buffer is full.

The standard calls BIO_ctrl_pending() and BIO_ctrl_wpending() can be used to determine the amount of pending data in the read or write buffer.

The BIO_reset() function clears any data in the write buffer.

The BIO_make_bio_pair() function joins two separate BIOs into a connected pair.

The BIO_destroy_pair() function destroys the association between two connected BIOs. Freeing up any half of the pair will automatically destroy the association.

The BIO_shutdown_wr() is used to close down a BIO b. After this call no further writes on BIO b are allowed. They will return an error. Reads on the other half of the pair will return any pending data or EOF when all pending data has been read.

The BIO_set_write_buf_size() function sets the write buffer size of BIO b to size. If the size is not initialized a default value is used. This is currently 17K, sufficient for a maximum size TLS record.

The BIO_get_write_buf_size() function returns the size of the write buffer.

The BIO_new_bio_pair() function combines the calls to BIO_new(), BIO_make_bio_pair(), and BIO_set_write_buf_size() to create a connected pair of BIOs, bio1 and bio2, with write buffer sizes writebuf1 and writebuf2. If either size is zero then the default size is used.

BIO_get_write_guarantee() and BIO_ctrl_get_write_guarantee() return the maximum length of data that can be written to the BIO. Writes larger than this value will return a value from BIO_write() less than the amount requested or, if the buffer is full request, a retry. BIO_ctrl_get_write_guarantee() is a function whereas BIO_get_write_guarantee() is a macro.

BIO_get_read_request() and BIO_ctrl_get_read_request() return the amount of data requested, or the buffer size if it is less, if the last read attempt at the other half of the BIO pair failed due to an empty buffer. This can be used to determine how much data should be written to the BIO so the next read will succeed. This is most useful in TLS/SSL applications where the amount of data read is usually meaningful rather than just a buffer size. After a successful read this call will return zero. It also will return zero once new data has been written satisfying the read request or part of it. BIO_get_read_request() never returns an amount larger than that returned by BIO_get_write_guarantee().

BIO_ctrl_reset_read_request() can also be used to reset the value returned by BIO_get_read_request() to zero.

NOTES

Both halves of a BIO pair should be freed. Even if one half is implicit freed due to a BIO_free_all() or SSL_free() call, the other half needs to be freed.

When used in bidirectional applications, such as TLS/SSL, care should be taken to flush any data in the write buffer. This can be done by calling BIO_pending() on the other half of the pair and, if any data is pending, reading it and sending it to the underlying transport. This must be done before any normal processing, such as calling select(), due to a request and BIO_should_read() being true.

To see why this is important consider a case where a request is sent using BIO_write() and a response read with BIO_read(), this can occur during an TLS/SSL handshake for example. BIO_write() will succeed and place data in the write buffer. BIO_read() will initially fail and BIO_should_read() will be true. If the application then waits for data to be available on the underlying transport before flushing the write buffer it will never succeed because the request was never sent.

BIO_s_connect

NAME

BIO_s_connect, BIO_set_conn_hostname, BIO_set_conn_port, BIO_set_conn_ip, BIO_set_conn_int_port, BIO_get_conn_hostname, BIO_get_conn_port, BIO_get_conn_ip, BIO_get_conn_int_port, BIO_set_nbio, BIO_do_connect – Connect BIO

SYNOPSIS

```
#include <openssl/bio.h>

BIO_METHOD * BIO_s_connect(
        void
);

#define BIO_set_conn_hostname(b,name) BIO_ctrl(b,BIO_C_SET_CONNECT,0,(char *)name)
#define BIO_set_conn_port(b,port) BIO_ctrl(b,BIO_C_SET_CONNECT,1,(char *)port)
#define BIO_set_conn_ip(b,ip)  BIO_ctrl(b,BIO_C_SET_CONNECT,2,(char *)ip)
#define BIO_set_conn_int_port(b,port) BIO_ctrl(b,BIO_C_SET_CONNECT,3,(char *)port)
#define BIO_get_conn_hostname(b)  BIO_ptr_ctrl(b,BIO_C_GET_CONNECT,0)
#define BIO_get_conn_port(b)      BIO_ptr_ctrl(b,BIO_C_GET_CONNECT,1)
#define BIO_get_conn_ip(b,ip) BIO_ptr_ctrl(b,BIO_C_SET_CONNECT,2)
#define BIO_get_conn_int_port(b,port) BIO_int_ctrl(b,BIO_C_SET_CONNECT,3,port)
#define BIO_set_nbio(b,n)BIO_ctrl(b,BIO_C_SET_NBIO,(n),NULL)
#define BIO_do_connect(b)BIO_do_handshake(b)
```

DESCRIPTION

The `BIO_s_connect()` function returns the connect BIO method. This is a wrapper round the platform's TCP/IP socket connection routines.

Using connect BIOs TCP/IP connections can be made and data transferred using only BIO routines. In this way any platform specific operations are hidden by the BIO abstraction.

Read and write operations on a connect BIO will perform I/O on the underlying connection. If no connection is established and the port and hostname is set up properly then a connection is established first.

Connect BIOs support `BIO_puts()` but not `BIO_gets()`.

If the close flag is set on a connect BIO then any active connection is shutdown and the socket closed when the BIO is freed.

Calling `BIO_reset()` on a connect BIO will close any active connection and reset the BIO into a state where it can connect to the same host again.

The `BIO_get_fd()` function places the underlying socket in `c` if it is not NULL. It also returns the socket . If `c` is not NULL it should be of type (int *).

`BIO_set_conn_hostname()` uses the string `name` to set the hostname. The hostname can be an IP address. The hostname can also include the port in the form `hostname:port`. It is also acceptable to use the form `hostname/any/other/path` or `hostname:port/any/other/path`.

The `BIO_set_conn_port()` function sets the port to `port`. The `port` can be the numerical form or a string such as `http`. A string will be looked up first using `getservbyname()` on the host platform, but if that fails a standard table of port names will be used. Currently the list is http, telnet, socks, https, ssl, ftp, gopher and wais.

The BIO_set_conn_ip() function sets the IP address to ip using binary form, that is four bytes specifying the IP address in big-endian form.

The BIO_set_conn_int_port() function sets the port using port. The port should be of type (int *).

The BIO_get_conn_hostname() function returns the hostname of the connect BIO or NULL if the BIO is initialized but no hostname is set. This return value is an internal pointer which should not be modified.

The BIO_get_conn_port() functon returns the port as a string.

The BIO_get_conn_ip() function returns the IP address in binary form.

The BIO_get_conn_int_port() function returns the port as an int.

The BIO_set_nbio() function sets the non blocking I/O flag to n. If n is zero then blocking I/O is set. If n is 1 then non blocking I/O is set. Blocking I/O is the default. The call to BIO_set_nbio() should be made before the connection is established because nonblocking I/O is set during the connect process.

The BIO_do_connect() function attempts to connect the supplied BIO. It returns 1 if the connection was established successfully. A zero or negative value is returned if the connection could not be established. The BIO_should_retry() function should be used for nonblocking connect BIOs to determine if the call should be retried.

NOTES

If blocking I/O is set, then a nonpositive return value from any I/O call is caused by an error condition. A zero return will normally mean that the connection was closed.

If the port name is supplied as part of the host name then this will override any value set with BIO_set_conn_port(). This might be undesirable if the application does not wish to allow connection to arbitrary ports. This can be avoided by checking for the presence of the colon (:) character in the passed hostname, and either indicating an error or truncating the string at that point.

The values returned by BIO_get_conn_hostname(), BIO_get_conn_port(), BIO_get_conn_ip(), and BIO_get_conn_int_port() are updated when a connection attempt is made. Before any connection attempt the values returned are those set by the application itself.

Applications do not have to call BIO_do_connect() but may wish to do so to separate the connection process from other I/O processing.

If nonblocking I/O is set then retries will be requested as appropriate.

It addition to BIO_should_read() and BIO_should_write() it is also possible for BIO_should_io_special() to be true during the initial connection process with the reason BIO_RR_CONNECT. If this is returned then this is an indication that a connection attempt would block. The application should then take appropriate action to wait until the underlying socket has connected and retry the call.

RETURN VALUES

BIO_s_connect() returns the connect BIO method.

BIO_get_fd() returns the socket or -1 if the BIO has not been initialized.

BIO_set_conn_hostname(), BIO_set_conn_port(), BIO_set_conn_ip(), and BIO_set_conn_int_port() always return 1.

BIO_get_conn_hostname() returns the connected hostname or NULL is none was set.

BIO_get_conn_port() returns a string representing the connected port or NULL if not set.

`BIO_get_conn_ip()` returns a pointer to the connected IP address in binary form or all zeros if not set.

`BIO_get_conn_int_port()` returns the connected port or 0 if none was set.

`BIO_set_nbio()` always returns 1.

`BIO_do_connect()` returns 1 if the connection was successfully established and 0 or -1 if the connection failed.

EXAMPLES

This example connects to a webserver on the local host and attempts to retrieve a page and copy the result to standard output.

```
BIO *cbio, *out;
int len;
char tmpbuf[1024];
ERR_load_crypto_strings();
cbio = BIO_new_connect("localhost:http");
out = BIO_new_fp(stdout, BIO_NOCLOSE);

if(BIO_do_connect(cbio) <= 0) {fprintf(stderr, "Error connecting to server\n");
ERR_print_errors_fp(stderr);

/* whatever ... */
}

BIO_puts(cbio, "GET / HTTP/1.0\n\n");
for(;;) {
len = BIO_read(cbio, tmpbuf, 1024);
if(len <= 0) break;
BIO_write(out, tmpbuf, len);
}

BIO_free(cbio);
BIO_free(out);
```

BIO_s_fd

NAME

BIO_s_fd, BIO_set_fd, BIO_get_fd, BIO_new_fd – File descriptor BIO

SYNOPSIS

```
#include <openssl/bio.h>

BIO_METHOD *BIO_s_fd(
        void
);

#define BIO_set_fd(b,fd,c)BIO_int_ctrl(b,BIO_C_SET_FD,c,fd)
#define BIO_get_fd(b,c)BIO_ctrl(b,BIO_C_GET_FD,0,(char *)c)

BIO *BIO_new_fd(
        int fd, int close_flag
);
```

DESCRIPTION

The `BIO_s_fd()` function returns the file descriptor BIO method. This is a wrapper around the platform's file descriptor routines, such as `read()` and `write()`.

The `BIO_read()` and `BIO_write()` functions read or write the underlying descriptor. `BIO_puts()` is supported, but `BIO_gets()` is not.

If the close flag is set then then `close()` is called on the underlying file descriptor when the BIO is freed.

The `BIO_reset()` function attempts to change the file pointer to the start of file using lseek(fd, 0, 0).

The `BIO_seek()` function sets the file pointer to position `ofs` from start of file using lseek(fd, ofs, 0).

The `BIO_tell()` function returns the current file position by calling lseek(fd, 0, 1).

The `BIO_set_fd()` function sets the file descriptor of BIO `b` to `fd` and the close flag to `c`.

The `BIO_get_fd()` function places the file descriptor in `c` if it is not NULL. It also returns the file descriptor. If `c` is not NULL it should be of type (int *).

The `BIO_new_fd()` function returns a file descriptor BIO using `fd` and `close_flag`.

NOTES

The behavior of the `BIO_read()` and `BIO_write()` functions depends on the behavior of the platform's `read()` and `write()` calls on the descriptor. If the underlying file descriptor is in a nonblocking mode then the BIO will behave in the manner described in the *BIO_read* and *BIO_should_retry* reference pages.

File descriptor BIOs should not be used for socket I/O. Use socket BIOs instead.

RETURN VALUES

`BIO_s_fd()` returns the file descriptor BIO method.

`BIO_reset()` returns zero for success and -1 if an error occurred. `BIO_seek()` and `BIO_tell()` return the current file position or -1 is an error occurred. These values reflect the underlying `lseek()` behavior.

`BIO_set_fd()` always returns 1.

`BIO_get_fd()` returns the file descriptor or -1 if the BIO has not been initialized.

`BIO_new_fd()` returns the newly allocated BIO or NULL is an error occurred.

EXAMPLES

This is a file descriptor BIO version of "Hello World":

```
BIO *out;
out = BIO_new_fd(fileno(stdout), BIO_NOCLOSE);
BIO_printf(out, "Hello World\n");
BIO_free(out);
```

SEE ALSO

Functions: *BIO_seek*, *BIO_tell*, *BIO_reset*, *BIO_read*, *BIO_write*, *BIO_puts*, *BIO_gets*, *BIO_printf*, *BIO_set_close*, *BIO_get_close*

BIO_s_file

NAME

BIO_s_file, BIO_new_file, BIO_new_fp, BIO_set_fp, BIO_get_fp, BIO_read_filename, BIO_write_filename, BIO_append_filename, BIO_rw_filename – FILE bio

SYNOPSIS

```
#include <openssl/bio.h>
BIO_METHOD *BIO_s_file(
        void
);
BIO *BIO_new_file(
       const char *filename, const char *mode
);
BIO *BIO_new_fp(
       FILE *stream, int flags
);
BIO_set_fp(
       BIO *b,FILE *fp, int flags
);
BIO_get_fp(
       BIO *b,FILE **fpp
);
int BIO_read_filename(
       BIO *b, char *name
);
int BIO_write_filename(
       BIO *b, char *name
);
int BIO_append_filename(
        BIO *b, char *name
);
int BIO_rw_filename(
       BIO *b, char *name
);
```

DESCRIPTION

The `BIO_s_file()` function returns the BIO file method. As its name implies it is a wrapper round the stdio FILE structure and it is a source/sink BIO.

Calls to `BIO_read()` and `BIO_write()` read and write data to the underlying stream. `BIO_gets()` and `BIO_puts()` are supported on file BIOs.

BIO_flush() on a file BIO calls the fflush() function on the wrapped stream.

BIO_reset() attempts to change the file pointer to the start of file using fseek(stream, 0, 0).

BIO_seek() sets the file pointer to position ofs from start of file using fseek(stream, ofs, 0).

BIO_eof() calls feof().

Setting the BIO_CLOSE flag calls fclose() on the stream when the BIO is freed.

BIO_new_file() creates a new file BIO with mode mode the meaning of mode is the same as the stdio function fopen(). The BIO_CLOSE flag is set on the returned BIO.

BIO_new_fp() creates a file BIO wrapping stream. Flags can be: BIO_CLOSE, BIO_NOCLOSE (the close flag) BIO_FP_TEXT (sets the underlying stream to text mode, default is binary: this only has any effect under Win32).

BIO_set_fp() sets the fp of a file BIO to fp. flags has the same meaning as in BIO_new_fp(), it is a macro.

BIO_get_fp() retrieves the fp of a file BIO, it is a macro.

BIO_seek() is a macro that sets the position pointer to offset bytes from the start of file.

BIO_tell() returns the value of the position pointer.

BIO_read_filename(), BIO_write_filename(), BIO_append_filename(), and BIO_rw_filename() set the file BIO b to use file namefor reading, writing, append or read write respectively.

NOTES

When wrapping stdout, stdin or stderr the underlying stream should not normally be closed. So the BIO_NOCLOSE flag should be set.

Because the file BIO calls the underlying stdio functions, any quirks in stdio behavior will be mirrored by the corresponding BIO.

RESTRICTIONS

BIO_reset() and BIO_seek() are implemented using fseek() on the underlying stream. The return value for fseek() is 0 for success or -1 if an error occurred this differs from other types of BIO which will typically return 1 for success and a nonpositive value if an error occurred.

RETURN VALUES

BIO_s_file() returns the file BIO method.

BIO_new_file() and BIO_new_fp() return a file BIO or NULL if an error occurred.

BIO_set_fp() and BIO_get_fp() return 1 for success or 0 for failure (although the current implementation never returns 0).

BIO_seek() returns the same value as the underlying fseek() function: 0 for success or -1 for failure.

BIO_tell() returns the current file position.

BIO_read_filename(), BIO_write_filename(), BIO_append_filename(), and BIO_rw_filename() return 1 for success or 0 for failure.

EXAMPLES

File BIO "hello world":

```
BIO *bio_out;
bio_out = BIO_new_fp(stdout, BIO_NOCLOSE);
BIO_printf(bio_out, "Hello World\n");
```

Alternative technique:

```
BIO *bio_out;
bio_out = BIO_new(BIO_s_file());
if(bio_out == NULL) /* Error ... */

if(!BIO_set_fp(bio_out, stdout, BIO_NOCLOSE)) /* Error ... */
BIO_printf(bio_out, "Hello World\n");
```

Write to a file:

```
BIO *out;
out = BIO_new_file("filename.txt", "w");
if(!out) /* Error occurred */
BIO_printf(out, "Hello World\n");
BIO_free(out);
```

Alternative technique:

```
BIO *out;
out = BIO_new(BIO_s_file());
if(out == NULL) /* Error ... */
if(!BIO_write_filename(out, "filename.txt")) /* Error ... */
BIO_printf(out, "Hello World\n");
BIO_free(out);
```

SEE ALSO

Functions: *BIO_seek*, *BIO_tell*, *BIO_reset*, *BIO_flush*, *BIO_read*, *BIO_write*, *BIO_puts*, *BIO_gets*, *BIO_printf*, *BIO_set_close*, *BIO_get_close*

BIO_s_mem

NAME

BIO_s_mem, BIO_set_mem_eof_return, BIO_get_mem_data, BIO_set_mem_buf, BIO_get_mem_ptr, BIO_new_mem_buf – Memory BIO

SYNOPSIS

```
#include <openssl/bio.h>

BIO_METHOD *BIO_s_mem(
        void
);
```

```
BIO_set_mem_eof_return(BIO *b,int v)
long BIO_get_mem_data(BIO *b, char **pp)
BIO_set_mem_buf(BIO *b,BUF_MEM *bm,int c)
BIO_get_mem_ptr(BIO *b,BUF_MEM **pp)
```

```
BIO *BIO_new_mem_buf(
        void *buf, int len
);
```

DESCRIPTION

`BIO_s_mem()` return the memory BIO method function.

A memory BIO is a source/sink BIO which uses memory for its I/O. Data written to a memory BIO is stored in a BUF_MEM structure which is extended as appropriate to accommodate the stored data.

Any data written to a memory BIO can be recalled by reading from it. Unless the memory BIO is read only any data read from it is deleted from the BIO.

Memory BIOs support `BIO_gets()` and `BIO_puts()`.

If the BIO_CLOSE flag is set when a memory BIO is freed then the underlying BUF_MEM structure is also freed.

Calling `BIO_reset()` on a read write memory BIO clears any data in it. On a read only BIO it restores the BIO to its original state and the read only data can be read again.

`BIO_eof()` is true if no data is in the BIO.

`BIO_ctrl_pending()` returns the number of bytes currently stored.

`BIO_set_mem_eof_return()` sets the behaviour of memory BIO `b` when it is empty. If the `v` is zero then an empty memory BIO will return EOF (that is it will return zero and BIO_should_retry(b) will be false. If `v` is non zero then it will return `v` when it is empty and it will set the read retry flag (that is BIO_read_retry(b) is true). To avoid ambiguity with a normal positive return value `v` should be set to a negative value, typically -1.

`BIO_get_mem_data()` sets `pp` to a pointer to the start of the memory BIOs data and returns the total amount of data available. It is implemented as a macro.

`BIO_set_mem_buf()` sets the internal BUF_MEM structure to `bm` and sets the close flag to `c`, that is `c` should be either BIO_CLOSE or BIO_NOCLOSE. It is a macro.

`BIO_get_mem_ptr()` places the underlying BUF_MEM structure in `pp`. It is a macro.

BIO_new_mem_buf() creates a memory BIO using len bytes of data at buf. If len is -1 then the buf is assumed to be null terminated and its length is determined by strlen. The BIO is set to a read only state and as a result cannot be written to. This is useful when some data needs to be made available from a static area of memory in the form of a BIO. The supplied data is read directly from the supplied buffer. It is not copied first. So the supplied area of memory must be unchanged until the BIO is freed.

NOTES

Writes to memory BIOs will always succeed if memory is available. Their size can grow indefinitely.

Every read from a read-write memory BIO will remove the data just read with an internal copy operation. If a BIO contains much data and it is read in small chunks, the operation can be very slow. The use of a read-only-memory BIO avoids this problem. If the BIO must be read-write, then adding a buffering BIO to the chain will speed up the process.

RESTRICTIONS

There should be an option to set the maximum size of a memory BIO.

There should be a way to rewind a read-write BIO without destroying its contents.

To improve efficiency, the copying operation should not occur after every small read of a large BIO.

EXAMPLES

Create a memory BIO and write some data to it:

```
BIO *mem = BIO_new(BIO_s_mem());
BIO_puts(mem, "Hello World\n");
```

Create a read only memory BIO:

```
char data[] = "Hello World";
BIO *mem;
mem = BIO_new_mem_buf(data, -1);
```

Extract the BUF_MEM structure from a memory BIO and then free up the BIO:

```
BUF_MEM *bptr;
BIO_get_mem_ptr(mem, &bptr);
BIO_set_close(mem, BIO_NOCLOSE); /* So BIO_free() leaves BUF_MEM alone */
BIO_free(mem);
```

BIO_s_null

NAME

BIO_s_null – Null data sink

SYNOPSIS

```
#include <openssl/bio.h>
BIO_METHOD *BIO_s_null(
       void
);
```

DESCRIPTION

`BIO_s_null()` returns the null sink BIO method. Data written to the null sink is discarded; reads return EOF.

NOTES

A null sink BIO behaves in a similar manner to the Unix /dev/null device.

A null bio can be placed on the end of a chain to discard any data passed through it.

A null sink is useful if, for example, an application wishes to digest some data by writing through a digest bio but not send the digested data anywhere. Since a BIO chain must normally include a source/sink BIO this can be achieved by adding a null sink BIO to the end of the chain

RETURN VALUES

`BIO_s_null()` returns the null sink BIO method.

BIO_s_socket

NAME

BIO_s_socket, BIO_new_socket – Socket BIO

SYNOPSIS

```
#include <openssl/bio.h>

BIO_METHOD *BIO_s_socket(
       void
);

#define BIO_set_fd(b,fd,c)BIO_int_ctrl(b,BIO_C_SET_FD,c,fd)
#define BIO_get_fd(b,c)BIO_ctrl(b,BIO_C_GET_FD,0,(char *)c)

BIO *BIO_new_socket(
       int sock, int close_flag
);
```

DESCRIPTION

The `BIO_s_socket()` function returns the socket BIO method. This is a wrapper around the platform's socket routines.

`BIO_read()` and `BIO_write()` read or write the underlying socket. `BIO_puts()` is supported, but `BIO_gets()` is not.

If the close flag is set then the socket is shut down and closed when the BIO is freed.

`BIO_set_fd()` sets the socket of BIO `b` to `fd` and the close flag to `c`.

`BIO_get_fd()` places the socket in `c` if it is not NULL, it also returns the socket . If `c` is not NULL it should be of type (int *).

`BIO_new_socket()` returns a socket BIO using `sock` and `close_flag`.

NOTES

Socket BIOs also support any relevant functionality of file descriptor BIOs.

The reason for having separate file descriptor and socket BIOs is that on some platforms sockets are not file descriptors and use distinct I/O routines, Windows® is one such platform. Any code mixing the two will not work on all platforms.

RETURN VALUES

`BIO_s_socket()` returns the socket BIO method.

`BIO_set_fd()` always returns 1.

`BIO_get_fd()` returns the socket or -1 if the BIO has not been initialized.

`BIO_new_socket()` returns the newly allocated BIO or NULL is an error occurred.

BIO_set_callback

NAME

BIO_set_callback, BIO_get_callback, BIO_set_callback_arg, BIO_get_callback_arg, BIO_debug_callback – BIO callback functions

SYNOPSIS

```
#include <openssl/bio.h>
#define BIO_set_callback(b,cb)((b)->callback=(cb))
#define BIO_get_callback(b)((b)->callback)
#define BIO_set_callback_arg(b,arg)((b)->cb_arg=(char *)(arg))
#define BIO_get_callback_arg(b)((b)->cb_arg)
```

```
long BIO_debug_callback(

       BIO *bio,int cmd,const char *argp,int argi, long argl,long ret

);
```

```
typedef long callback(

       BIO *b, int oper, const char *argp, int argi, long argl, long retvalue

);
```

DESCRIPTION

`BIO_set_callback()` and `BIO_get_callback()` set and retrieve the BIO callback, they are both macros. The callback is called during most high level BIO operations. It can be used for debugging purposes to trace operations on a BIO or to modify its operation.

`BIO_set_callback_arg()` and `BIO_get_callback_arg()` are macros which can be used to set and retrieve an argument for use in the callback.

`BIO_debug_callback()` is a standard debugging callback which prints out information relating to each BIO operation. If the callback argument is set it is interpreted as a BIO to send the information to, otherwise stderr is used.

The `callback()` is the callback function itself. The meaning of each argument is described below.

The BIO that the callback is attached to is passed in `b`.

The `oper` is set to the operation being performed. For some operations the callback is called twice, once before and once after the actual operation, the latter case has `oper` or'ed with BIO_CB_RETURN.

The meaning of the arguments `argp`, `argi`, and `argl` depends on the value of `oper`, that is the operation being performed.

The `retvalue` is the return value that would be returned to the application if no callback were present. The actual value returned is the return value of the callback itself. In the case of callbacks called before the actual BIO operation 1 is placed in `retvalue`. If the return value is not positive it will be returned to the application, and the BIO operation will not be performed.

The callback should return `retvalue` when it finishes processing, unless it specifically wishes to modify the value returned to the application.

Callback Operations

`BIO_free(b)`

callback(b, BIO_CB_FREE, NULL, 0L, 0L, 1L) is called before the free operation.

`BIO_read(b, out, outl)`

callback(b, BIO_CB_READ, out, outl, 0L, 1L) is called before the read and callback(b, BIO_CB_READ|BIO_CB_RETURN, out, outl, 0L, retvalue) after.

`BIO_write(b, in, inl)`

callback(b, BIO_CB_WRITE, in, inl, 0L, 1L) is called before the write and callback(b, BIO_CB_WRITE|BIO_CB_RETURN, in, inl, 0L, retvalue) after.

`BIO_gets(b, out, outl)`

callback(b, BIO_CB_GETS, out, outl, 0L, 1L) is called before the operation and callback(b, BIO_CB_GETS|BIO_CB_RETURN, out, outl, 0L, retvalue) after.

`BIO_puts(b, in)`

callback(b, BIO_CB_WRITE, in, 0, 0L, 1L) is called before the operation and callback(b, BIO_CB_WRITE|BIO_CB_RETURN, in, 0, 0L, retvalue) after.

`BIO_ctrl(BIO *b, int cmd, long larg, void *parg)`

callback(b,BIO_CB_CTRL,parg,cmd,larg,1L) is called before the call and callback(b,BIO_CB_CTRL|BIO_CB_RETURN,parg,cmd, larg,ret) after.

EXAMPLES

The `BIO_debug_callback()` function is a good example. Its source is in `crypto/bio/bio_cb.c`

BIO_should_retry

NAME

BIO_should_retry, BIO_should_read, BIO_should_write, BIO_should_io_special, BIO_retry_type, BIO_get_retry_BIO, BIO_get_retry_reason – BIO retry functions

SYNOPSIS

```
#include <openssl/bio.h>
#define BIO_should_read(a)((a)->flags & BIO_FLAGS_READ)
#define BIO_should_write(a)((a)->flags & BIO_FLAGS_WRITE)
#define BIO_should_io_special(a)((a)->flags & BIO_FLAGS_IO_SPECIAL)
#define BIO_retry_type(a)((a)->flags & BIO_FLAGS_RWS)
#define BIO_should_retry(a)((a)->flags & BIO_FLAGS_SHOULD_RETRY)
#define BIO_FLAGS_READ0x01
#define BIO_FLAGS_WRITE0x02
#define BIO_FLAGS_IO_SPECIAL0x04
#define BIO_FLAGS_RWS (BIO_FLAGS_READ|BIO_FLAGS_WRITE|BIO_FLAGS_IO_SPECIAL)
#define BIO_FLAGS_SHOULD_RETRY0x08

BIO * BIO_get_retry_BIO(

       BIO *bio, int *reason

);

int BIO_get_retry_reason(

       BIO *bio

);
```

DESCRIPTION

These functions determine why a BIO is not able to read or write data. They will typically be called after a failed `BIO_read()` or `BIO_write()` call.

The `BIO_should_retry()` function is true if the call that produced this condition should then be retried at a later time.

If `BIO_should_retry()` is false then the cause is an error condition.

The `BIO_should_read()` function is true if the cause of the condition is that a BIO needs to read data.

The `BIO_should_write()` function is true if the cause of the condition is that a BIO needs to read data.

The `BIO_should_io_special()` function is true if some special condition, other than reading or writing, is the cause of the condition.

The `BIO_get_retry_reason()` function returns a mask of the cause of a retry condition consisting of the values `BIO_FLAGS_READ`, `BIO_FLAGS_WRITE`, and `BIO_FLAGS_IO_SPECIAL`. Current BIO types will only set one of these.

The `BIO_get_retry_BIO()` function determines the precise reason for the special condition. It returns the BIO that caused this condition and if `reason` is not NULL it contains the reason code. The meaning of the reason code and the action that should be taken depends on the type of BIO that resulted in this condition.

The `BIO_get_retry_reason()` function returns the reason for a special condition if passed the relevant BIO, for example as returned by `BIO_get_retry_BIO()`.

NOTES

If `BIO_should_retry()` returns false then the precise error condition depends on the BIO type that caused it and the return code of the BIO operation. For example, if a call to `BIO_read()` on a socket BIO returns 0 and `BIO_should_retry()` is false then the cause will be that the connection closed. A similar condition on a file BIO will mean that it has reached EOF. Some BIO types may place additional information on the error queue. For more details see the individual BIO type manual pages.

If the underlying I/O structure is in a blocking mode almost all current BIO types will not request a retry, because the underlying I/O calls will not. If the application knows that the BIO type will never signal a retry then it need not call `BIO_should_retry()` after a failed BIO I/O call. This is typically done with file BIOs.

SSL BIOs are the only current exception to this rule. They can request a retry even if the underlying I/O structure is blocking, if a handshake occurs during a call to `BIO_read()`. An application can retry the failed call immediately or avoid this situation by setting SSL_MODE_AUTO_RETRY on the underlying SSL structure.

While an application can retry a failed nonblocking call immediately, this is likely to be very inefficient because the call will fail repeatedly until data can be processed or is available. An application will normally wait until the necessary condition is satisfied. How this is done depends on the underlying I/O structure.

For example, if the cause is ultimately a socket and `BIO_should_read()` is true then a call to `select()` may be made to wait until data is available and then retry the BIO operation. By combining the retry conditions of several non blocking BIOs in a single `select()` call it is possible to service several BIOs in a single thread, though the performance may be poor if SSL BIOs are present because long delays can occur during the initial handshake process.

It is possible for a BIO to block indefinitely if the underlying I/O structure cannot process or return any data. This depends on the behavior of the platforms I/O functions. This is often not desirable. One solution is to use nonblocking I/O and use a timeout on the `select()` (or equivalent) call.

RESTRICTIONS

The OpenSSL ASN1 functions cannot gracefully deal with nonblocking I/O. They cannot retry after a partial read or write. This is usually worked around by only passing the relevant data to ASN1 functions when the entire structure can be read or written.

blowfish

NAME

blowfish, BF_set_key, BF_encrypt, BF_decrypt, BF_ecb_encrypt, BF_cbc_encrypt, BF_cfb64_encryp, BF_ofb64_encrypt, BF_options – Blowfish encryption

SYNOPSIS

```
#include <openssl/blowfish.h>

void BF_set_key(
        BF_KEY *key, int len, const unsigned char *data
);

void BF_ecb_encrypt(
        const unsigned char *in, unsigned char *out, BF_KEY *key, int enc
);

void BF_cbc_encrypt(
        const unsigned char *in, unsigned char *out, long length, BF_KEY *schedule,
        unsigned char *ivec, int enc
);

void BF_cfb64_encrypt(
        const unsigned char *in, unsigned char *out, long length, BF_KEY *schedule,
        unsigned char *ivec, int *num, int enc
);

void BF_ofb64_encrypt(
        const unsigned char *in, unsigned char *out, long length, BF_KEY *schedule,
        unsigned char *ivec, int *num
);

const char *BF_options(
        void
);

void BF_encrypt(
        BF_LONG *data,const BF_KEY *key
);

void BF_decrypt(
        BF_LONG *data,const BF_KEY *key
);
```

DESCRIPTION

This library implements the Blowfish cipher, which is invented and described by Counterpane. See http://www.counterpane.com/blowfish.html.

Blowfish is a block cipher that operates on 64-bit (8 byte) blocks of data. It uses a variable size key, but typically, 128-bit (16 byte) keys are considered good for strong encryption. Blowfish can be used in the same modes as DES. See *des_modes*. Blowfish is one of the faster block ciphers. It is faster than DES, and much faster than IDEA or RC2.

Blowfish consists of a key setup phase and the actual encryption or decryption phase.

The `BF_set_key()` function sets up the `BF_KEYkey` using the `len` bytes long key at `data`.

The `BF_ecb_encrypt()` function is the basic Blowfish encryption and decryption function. It encrypts or decrypts the first 64 bits of `in` using the key `key`, putting the result in `out`. The `enc` decides if encryption (`BF_ENCRYPT`) or decryption (`BF_DECRYPT`) shall be performed. The vector pointed at by `in` and `out` must be 64 bits in length, no less. If they are larger, everything after the first 64 bits is ignored.

The `BF_cbc_encrypt()`, `BF_cfb64_encrypt()`, and `BF_ofb64_encrypt()` mode functions all operate on variable length data. They all take an initialization vector `ivec` which needs to be passed along into the next call of the same function for the same message. The `ivec` may be initialized with anything, but the recipient needs to know what was used, or it won't be able to decrypt. Some programs and protocols simplify this. For example, `ivec` is simply initialized to zero in SSH. The `BF_cbc_encrypt()` function operates of data that is a multiple of 8 bytes long, while the `BF_cfb64_encrypt()` and `BF_ofb64_encrypt()` functions are used to encrypt a variable number of bytes (the amount does not have to be an exact multiple of 8). The purpose of the latter two is to simulate stream ciphers, and therefore, they need the parameter `num`, which is a pointer to an integer where the current offset in `ivec` is stored between calls. This integer must be initialized to zero when `ivec` is initialized.

The `BF_cbc_encrypt()` function is the Cipher Block Chaining function for Blowfish. It encrypts or decrypts the 64 bits chunks of `in` using the key `schedule`, putting the result in `out`. The `enc` decides if encryption (BF_ENCRYPT) or decryption (BF_DECRYPT) shall be performed. The `ivec` must point at an 8 byte long initialization vector.

The `BF_cfb64_encrypt()` function is the CFB mode for Blowfish with 64-bit feedback. It encrypts or decrypts the bytes in `in` using the key `schedule`, putting the result in `out`. The `enc` decides if encryption (BF_ENCRYPT) or decryption (BF_DECRYPT) shall be performed. The `ivec` must point at an 8 byte long initialization vector. The `num` must point at an integer which must be initially zero.

The `BF_ofb64_encrypt()` function is the OFB mode for Blowfish with 64-bit feedback. It uses the same parameters as the `BF_cfb64_encrypt()` function, which must be initialized the same way.

The `BF_encrypt()` and `BF_decrypt()` functions are the lowest level functions for Blowfish encryption. They encrypt or decrypt the first 64 bits of the vector pointed by `data`, using the key `key`. They also take each 32-bit chunk in host-byte order, which is little-endian on little-endian platforms and big-endian on big-endian platforms. These functions should not be used unless you implement modes of Blowfish. The alternative is to use the `BF_ecb_encrypt()` function.

NOTES

Applications should use the higher level functions, such as `EVP_EncryptInit()`, instead of calling the blowfish functions directly.

RETURN VALUES

None of the functions presented here return any value.

HISTORY

The blowfish functions are available in all versions of SSLeay and OpenSSL.

bn

NAME

bn – Multiprecision integer arithmetics

SYNOPSIS

```
#include <openssl/bn.h>
BIGNUM *BN_new(
        void
);
void BN_free(
        BIGNUM *a
);
void BN_init(
        BIGNUM *
);
void BN_clear(
        BIGNUM *a
);
void BN_clear_free(
        BIGNUM *a
);
BN_CTX *BN_CTX_new(
        void
);
void BN_CTX_init(
        BN_CTX *c
);
void BN_CTX_free(
        BN_CTX *c
);
BIGNUM *BN_copy(
        BIGNUM *a, const BIGNUM *b
);
BIGNUM *BN_dup(
        const BIGNUM *a
);
int BN_num_bytes(
        const BIGNUM *a
);
```

```
int BN_num_bits(
        const BIGNUM *a
);
int BN_num_bits_word(
        BN_ULONG w
);
int BN_add(
        BIGNUM *r, BIGNUM *a, BIGNUM *b
);
int BN_sub(
        BIGNUM *r, const BIGNUM *a, const BIGNUM *b
);
int BN_mul(
        BIGNUM *r, BIGNUM *a, BIGNUM *b, BN_CTX *ctx
);
int BN_div(
        BIGNUM *dv, BIGNUM *rem, const BIGNUM *a, const BIGNUM *d, BN_CTX *ctx
);
int BN_sqr(
        BIGNUM *r, BIGNUM *a, BN_CTX *ctx
);
int BN_mod(
        BIGNUM *rem, const BIGNUM *a, const BIGNUM *m, BN_CTX *ctx
);
int BN_mod_mul(
        BIGNUM *ret, BIGNUM *a, BIGNUM *b, const BIGNUM *m, BN_CTX *ctx
);
int BN_exp(
        BIGNUM *r, BIGNUM *a, BIGNUM *p, BN_CTX *ctx
);
int BN_mod_exp(
        BIGNUM *r, BIGNUM *a, const BIGNUM *p, const BIGNUM *m, BN_CTX *ctx
);
int BN_gcd(
        BIGNUM *r, BIGNUM *a, BIGNUM *b, BN_CTX *ctx
);
int BN_add_word(
        BIGNUM *a, BN_ULONG w
);
int BN_sub_word(
```

```
        BIGNUM *a, BN_ULONG w
);
int BN_mul_word(
        BIGNUM *a, BN_ULONG w
);
BN_ULONG BN_div_word(
        BIGNUM *a, BN_ULONG w
);
BN_ULONG BN_mod_word(
        const BIGNUM *a, BN_ULONG w
);
int BN_cmp(
        BIGNUM *a, BIGNUM *b
);
int BN_ucmp(
        BIGNUM *a, BIGNUM *b
);
int BN_is_zero(
        BIGNUM *a
);
int BN_is_one(
        BIGNUM *a
);
int BN_is_word(
        BIGNUM *a, BN_ULONG w
);
int BN_is_odd(
        BIGNUM *a
);
int BN_zero(
        BIGNUM *a
);
int BN_one(
        BIGNUM *a
);
BIGNUM *BN_value_one(
        void
);
int BN_set_word(
        BIGNUM *a, unsigned long w
```

```
);
unsigned long BN_get_word(
         BIGNUM *a
);
int BN_rand(
        BIGNUM *rnd, int bits, int top, int bottom
);
int BN_pseudo_rand(
        BIGNUM *rnd, int bits, int top, int bottom
);
int BN_rand_range(
        BIGNUM *rnd, BIGNUM *range
);
BIGNUM *BN_generate_prime(
         BIGNUM *ret, int bits,int safe, BIGNUM *add, BIGNUM *rem, void (*callback)(int,
        int, void *), void *cb_arg
);
int BN_is_prime(
        const BIGNUM *p, int nchecks, void (*callback)(int, int, void *), BN_CTX *ctx,
        void *cb_arg
);
int BN_set_bit(
        BIGNUM *a, int n
);
int BN_clear_bit(
        BIGNUM *a, int n
);
int BN_is_bit_set(
        const BIGNUM *a, int n
);
int BN_mask_bits(
        BIGNUM *a, int n
);
int BN_lshift(
        BIGNUM *r, const BIGNUM *a, int n
);
int BN_lshift1(
        BIGNUM *r, BIGNUM *a
);
int BN_rshift(
        BIGNUM *r, BIGNUM *a, int n
```

```
);
int BN_rshift1(
        BIGNUM *r, BIGNUM *a
);
int BN_bn2bin(
        const BIGNUM *a, unsigned char *to
);
BIGNUM *BN_bin2bn(
        const unsigned char *s, int len, BIGNUM *ret
);
char *BN_bn2hex(
        const BIGNUM *a
);
char *BN_bn2dec(
        const BIGNUM *a
);
int BN_hex2bn(
        BIGNUM **a, const char *str
);
int BN_dec2bn(
        BIGNUM **a, const char *str
);
int BN_print(
        BIO *fp, const BIGNUM *a
);
int BN_print_fp(
        FILE *fp, const BIGNUM *a
);
int BN_bn2mpi(
        const BIGNUM *a, unsigned char *to
);
BIGNUM *BN_mpi2bn(
        unsigned char *s, int len, BIGNUM *ret
);
BIGNUM *BN_mod_inverse(
        BIGNUM *r, BIGNUM *a, const BIGNUM *n, BN_CTX *ctx
);
BN_RECP_CTX *BN_RECP_CTX_new(
         void
);
```

```
void BN_RECP_CTX_init(
        BN_RECP_CTX *recp
);
void BN_RECP_CTX_free(
        BN_RECP_CTX *recp
);
int BN_RECP_CTX_set(
        BN_RECP_CTX *recp, const BIGNUM *m, BN_CTX *ctx
);
int BN_mod_mul_reciprocal(
         BIGNUM *r, BIGNUM *a, BIGNUM *b, BN_RECP_CTX *recp, BN_CTX *ctx
);
BN_MONT_CTX *BN_MONT_CTX_new(
         void
);
void BN_MONT_CTX_init(
        BN_MONT_CTX *ctx
);
void BN_MONT_CTX_free(
        BN_MONT_CTX *mont
);
int BN_MONT_CTX_set(
        BN_MONT_CTX *mont, const BIGNUM *m, BN_CTX *ctx
);
BN_MONT_CTX *BN_MONT_CTX_copy(
         BN_MONT_CTX *to, BN_MONT_CTX *from
);
int BN_mod_mul_montgomery(
         BIGNUM *r, BIGNUM *a, BIGNUM *b, BN_MONT_CTX *mont, BN_CTX *ctx
);
int BN_from_montgomery(
        BIGNUM *r, BIGNUM *a, BN_MONT_CTX *mont, BN_CTX *ctx
);
int BN_to_montgomery(
        BIGNUM *r, BIGNUM *a, BN_MONT_CTX *mont, BN_CTX *ctx
);
```

DESCRIPTION

This library performs arithmetic operations on integers of arbitrary size. It was written for use in public key cryptography, such as RSA and Diffie-Hellman.

It uses dynamic memory allocation for storing its data structures. That means that there is no limit on the size of the numbers manipulated by these functions, but return values must always be checked in case a memory allocation error has occurred.

The basic object in this library is a BIGNUM. It is used to hold a single large integer. This type should be considered opaque and fields should not be modified or accessed directly.

The creation of BIGNUM objects is described in *BN_new*; *BN_add* describes most of the arithmetic operations. Comparison is described in *BN_cmp*; *BN_zero* describes certain assignments, *BN_rand* the generation of random numbers, *BN_generate_prime* deals with prime numbers and *BN_set_bit* with bit operations. The conversion of BIGNUMs to external formats is described in *BN_bn2bin*.

SEE ALSO

Functions: *bn_internal*, *dh*, *err*, *rand*, *rsa*, *BN_new*, *BN_CTX_new*, *BN_copy*, *BN_num_bytes*, *BN_add*, *BN_add_word*, *BN_cmp*, *BN_zero*, *BN_rand*, *BN_generate_prime*, *BN_set_bit*, *BN_bn2bin*, *BN_mod_inverse*, *BN_mod_mul_reciprocal*, *BN_mod_mul_montgomery*

BN_add

NAME

BN_add, BN_sub, BN_mul, BN_div, BN_sqr, BN_mod, BN_mod_mul, BN_exp , BN_mod_exp, BN_gcd – Arithmetic operations on BIGNUMs

SYNOPSIS

```
#include <openssl/bn.h>
int BN_add(
        BIGNUM *r, const BIGNUM *a, const BIGNUM *b
);
int BN_sub(
        BIGNUM *r, const BIGNUM *a, const BIGNUM *b
);
int BN_mul(
        BIGNUM *r, BIGNUM *a, BIGNUM *b, BN_CTX *ctx
);
int BN_div(
        BIGNUM *dv, BIGNUM *rem, const BIGNUM *a, const BIGNUM *d, BN_CTX *ctx
);
int BN_sqr(
        BIGNUM *r, BIGNUM *a, BN_CTX *ctx
);
int BN_mod(
        BIGNUM *rem, const BIGNUM *a, const BIGNUM *m, BN_CTX *ctx
);
int BN_mod_mul(
        BIGNUM *ret, BIGNUM *a, BIGNUM *b, const BIGNUM *m, BN_CTX *ctx
);
int BN_exp(
        BIGNUM *r, BIGNUM *a, BIGNUM *p, BN_CTX *ctx
);
int BN_mod_exp(
        BIGNUM *r, BIGNUM *a, const BIGNUM *p, const BIGNUM *m, BN_CTX *ctx
);
int BN_gcd(
        BIGNUM *r, BIGNUM *a, BIGNUM *b, BN_CTX *ctx
);
```

DESCRIPTION

The `BN_add()` function adds `a` and `b` and places the result in `r` (`r=a+b`). The `r` value can be the same `BIGNUM` as `a` or `b`.

The `BN_sub()` function subtracts `b` from `a` and places the result in `r` (`r=a-b`).

The `BN_mul()` function multiplies `a` and `b` and places the result in `r` (`r=a*b`). The `r` value may be the same `BIGNUM` as `a` or `b`. For multiplication by powers of 2, use the `BN_lshift()` function.

The `BN_div()` function divides `a` by `d` and places the result in `dv` and the remainder in `rem` (`dv=a/d`, `rem=a%d`). Either of `dv` and `rem` may be NULL, in which case the respective value is not returned. For division by powers of 2, use the `BN_rshift()` function.

The `BN_sqr()` function takes the square of `a` and places the result in `r` (`r=a^2`). The `r` and `a` values may be the same `BIGNUM`. This function is faster than `BN_mul(r,a,a)`.

The `BN_mod()` function finds the remainder of `a` divided by `m` and places it in `rem` (`rem=a%m`).

The `BN_mod_mul()` function multiplies `a` by `b` and finds the remainder when divided by `m` (`r=(a*b)%m`). The `r` value may be the same `BIGNUM` as `a` or `b`. For a more efficient algorithm, see *BN_mod_mul_montgomery*; for repeated computations using the same modulus, see *BN_mod_mul_reciprocal*.

The `BN_exp()` function raises `a` to the `p` power and places the result in `r` (`r=a^p`). This function is faster than repeated applications of `BN_mul()`.

`BN_mod_exp()` computes `a` to the `p` power modulo `m` (`r=a^p % m`). This function uses less time and space than `BN_exp()`.

`BN_gcd()` computes the greatest common divisor of `a` and `b` and places the result in `r`. The `r` value may be the same `BIGNUM` as `a` or `b`.

For all functions, `ctx` is a previously allocated `BN_CTX` used for temporary variables. See *BN_CTX_new*.

Unless noted otherwise, the result `BIGNUM` must be different from the arguments.

RETURN VALUES

For all functions, 1 is returned for success, 0 on error. The return value should always be checked (e.g., `if (!BN_add(r,a,b)) goto err;`). The error codes can be obtained by using the `ERR_get_error()` function.

HISTORY

The `BN_add()`, `BN_sub()`, `BN_div()`, `BN_sqr()`, `BN_mod()`, `BN_mod_mul()`, `BN_mod_exp()`, and `BN_gcd()` are available in all versions of SSLeay and OpenSSL. The `ctx` argument to `BN_mul()` was added in SSLeay 0.9.1b. The `BN_exp()` function appeared in SSLeay 0.9.0.

SEE ALSO

Functions: *bn*, *err*, *BN_CTX_new*, *BN_add_word*, *BN_set_bit*

BN_add_word

NAME

BN_add_word, BN_sub_word, BN_mul_word , BN_div_word, BN_mod_word – Arithmetic functions on BIGNUMs with integers

SYNOPSIS

```
#include <openssl/bn.h>

int BN_add_word(
        BIGNUM *a, BN_ULONG w
);

int BN_sub_word(
        BIGNUM *a, BN_ULONG w
);

int BN_mul_word(
        BIGNUM *a, BN_ULONG w
);

BN_ULONG BN_div_word(
        BIGNUM *a, BN_ULONG w
);

BN_ULONG BN_mod_word(
        const BIGNUM *a, BN_ULONG w
);
```

DESCRIPTION

These functions perform arithmetic operations on BIGNUMs with unsigned integers. They are much more efficient than the normal BIGNUM arithmetic operations.

The `BN_add_word()` function adds `w` to `a` (`a+=w`).

The `BN_sub_word()` function subtracts `w` from `a` (`a-=w`).

The `BN_mul_word()` function multiplies `a` and `w` (`a*=b`).

The `BN_div_word()` function divides `a` by `w` (`a/=w`) and returns the remainder.

The `BN_mod_word()` function returns the remainder of `a` divided by `w` (`a%m`).

For the `BN_div_word()` and `BN_mod_word()` functions, the value of `w` must not be 0.

RETURN VALUES

The `BN_add_word()`, `BN_sub_word()`, and `BN_mul_word()` functions return 1 for success, 0 on error. The error codes can be obtained by using the `ERR_get_error()` function.

The `BN_mod_word()` and `BN_div_word()` functions return `a%w`.

HISTORY

The `BN_add_word()` and `BN_mod_word()` functions are available in all versions of SSLeay and OpenSSL. The `BN_div_word()` function was added in SSLeay 0.8, and `BN_sub_word()` and `BN_mul_word()` in SSLeay 0.9.0.

SEE ALSO

Functions: *bn*, *err*, *BN_add*

BN_bn2bin

NAME

BN_bn2bin, BN_bin2bn, BN_bn2hex, BN_bn2dec, BN_hex2bn, BN_dec2bn, BN_print, BN_print_fp, BN_bn2mpi, BN_mpi2bn – Format conversions

SYNOPSIS

```
#include <openssl/bn.h>

int BN_bn2bin(
        const BIGNUM *a, unsigned char *to
);

BIGNUM *BN_bin2bn(
        const unsigned char *s, int len, BIGNUM *ret
);

char *BN_bn2hex(
        const BIGNUM *a
);

char *BN_bn2dec(
        const BIGNUM *a
);

int BN_hex2bn(
        BIGNUM **a, const char *str
);

int BN_dec2bn(
        BIGNUM **a, const char *str
);

int BN_print(
        BIO *fp, const BIGNUM *a
);

int BN_print_fp(
        FILE *fp, const BIGNUM *a
);

int BN_bn2mpi(
        const BIGNUM *a, unsigned char *to
);

BIGNUM *BN_mpi2bn(
        unsigned char *s, int len, BIGNUM *ret
);
```

DESCRIPTION

The BN_bn2bin() function converts the absolute value of a into big-endian form and stores it at to. The to value must point to BN_num_bytes(a) bytes of memory.

The BN_bin2bn() function converts the positive integer in big-endian form of length len at s into a BIGNUM and places it in ret. If ret is NULL, a new BIGNUM is created.

The BN_bn2hex() and BN_bn2dec() functions return printable strings containing the hexadecimal and decimal encoding of a respectively. For negative numbers, the string is prefaced with a leading '-'. The string must be freed later using OPENSSL_free().

BN_hex2bn() converts the string str containing a hexadecimal number to a BIGNUM and stores it in **bn. If *bn is NULL, a new BIGNUM is created. If bn is NULL, it only computes the number's length in hexadecimal digits. If the string starts with '-', the number is negative. BN_dec2bn() is the same using the decimal system.

The BN_print() and BN_print_fp() functions write the hexadecimal encoding of a, with a leading '-' for negative numbers, to the BIO or FILE fp.

The BN_bn2mpi() and BN_mpi2bn() functions convert BIGNUMs from and to a format that consists of the number's length in bytes represented as a 3-byte big-endian number, and the number itself in big-endian format, where the most significant bit signals a negative number (the representation of numbers with the MSB set is prefixed with null byte).

The BN_bn2mpi() function stores the representation of a at to, where to must be large enough to hold the result. The size can be determined by calling BN_bn2mpi(a, NULL).

The BN_mpi2bn() function converts the len bytes long representation at s to a BIGNUM and stores it at ret, or in a newly allocated BIGNUM if ret is NULL.

RETURN VALUES

The BN_bn2bin() function returns the length of the big-endian number placed at to. The BN_bin2bn() function returns the BIGNUM, NULL on error.

The BN_bn2hex() and BN_bn2dec() functions return a null-terminated string, or NULL on error. The BN_hex2bn() and BN_dec2bn() functions return the number's length in hexadecimal or decimal digits, and 0 on error.

The BN_print_fp() and BN_print() functions return 1 on success, 0 on write errors.

The BN_bn2mpi() function returns the length of the representation. The BN_mpi2bn() function returns the BIGNUM, and NULL on error.

The error codes can be obtained by using the ERR_get_error() function.

HISTORY

The BN_bn2bin(), BN_bin2bn(), BN_print_fp(), and BN_print() functions are available in all versions of SSLeay and OpenSSL.

The BN_bn2hex(), BN_bn2dec(), BN_hex2bn(), BN_dec2bn(), BN_bn2mpi(), and BN_mpi2bn() functions were added in SSLeay 0.9.0.

SEE ALSO

Functions: *bn*, *err*, *BN_zero*, *ASN1_INTEGER_to_BN*, *BN_num_bytes*

BN_cmp

NAME

BN_cmp, BN_ucmp, BN_is_zero, BN_is_one, BN_is_word, BN_is_odd – BIGNUM comparison and test functions

SYNOPSIS

```
#include <openssl/bn.h>
int BN_cmp(
       BIGNUM *a, BIGNUM *b
);
int BN_ucmp(
       BIGNUM *a, BIGNUM *b
);
int BN_is_zero(
       BIGNUM *a
);
int BN_is_one(
       BIGNUM *a
);
int BN_is_word(
       BIGNUM *a, BN_ULONG w
);
int BN_is_odd(
       BIGNUM *a
);
```

DESCRIPTION

The `BN_cmp()` function compares the numbers `a` and `b`. The `BN_ucmp()` function compares their absolute values.

The `BN_is_zero()`, `BN_is_one()`, and `BN_is_word()` test if `a` equals 0, 1, or `w` respectively. The `BN_is_odd()` tests if `a` is odd.

The `BN_is_zero()`, `BN_is_one()`, `BN_is_word()`, and `BN_is_odd()` are macros.

RETURN VALUES

The `BN_cmp()` functon returns -1 if `a` < `b`, 0 if `a` == `b` and 1 if `a` > `b`. The `BN_ucmp()` function is the same using the absolute values of `a` and `b`.

The `BN_is_zero()`, `BN_is_one()`, `BN_is_word()`, and `BN_is_odd()` macros return 1 if the condition is true, 0 otherwise.

HISTORY

The `BN_cmp()`, `BN_ucmp()`, `BN_is_zero()`, `BN_is_one()`, and `BN_is_word()` functions are available in all versions of SSLeay and OpenSSL. The `BN_is_odd()` function was added in SSLeay 0.8.

SEE ALSO

Functions: *bn*

BN_copy

NAME

BN_copy, BN_dup – Copy BIGNUMs

SYNOPSIS

```
#include <openssl/bn.h>
BIGNUM *BN_copy(
        BIGNUM *to, const BIGNUM *from
);
BIGNUM *BN_dup(
        const BIGNUM *from
);
```

DESCRIPTION

The `BN_copy()` function copies `from` to `to`. The `BN_dup()` function creates a new `BIGNUM` containing the value `from`.

RETURN VALUES

`BN_copy()` function returns `to` on success, NULL on error. The `BN_dup()` function returns the new `BIGNUM`, and NULL on error. The error codes can be obtained by using the `ERR_get_error()` function.

HISTORY

The `BN_copy()` and `BN_dup()` functions are available in all versions of SSLeay and OpenSSL.

SEE ALSO

Functions: *bn*, *err*

BN_CTX_new

NAME

BN_CTX_new, BN_CTX_init, BN_CTX_free – Allocate and free BN_CTX structures

SYNOPSIS

```
#include <openssl/bn.h>
BN_CTX *BN_CTX_new(
        void
);
void BN_CTX_init(
        BN_CTX *c
);
void BN_CTX_free(
        BN_CTX *c
);
```

DESCRIPTION

A `BN_CTX` is a structure that holds `BIGNUM` temporary variables used by library functions. Since dynamic memory allocation to create `BIGNUM`s is rather expensive when used in conjunction with repeated subroutine calls, the `BN_CTX` structure is used.

The `BN_CTX_new()` function allocates and initializes a `BN_CTX` structure. The `BN_CTX_init()` function initializes an existing uninitialized `BN_CTX`.

The `BN_CTX_free()` function frees the components of the `BN_CTX`, and if it was created by `BN_CTX_new()`, also the structure itself. If `BN_CTX_start()` has been used on the `BN_CTX`, `BN_CTX_end()` must be called before the `BN_CTX` may be freed by `BN_CTX_free()`.

RETURN VALUES

`BN_CTX_new()` returns a pointer to the `BN_CTX`. If the allocation fails, it returns `NULL` and sets an error code that can be obtained by `ERR_get_error()`.

The `BN_CTX_init()` and `BN_CTX_free()` functions have no return values.

HISTORY

The `BN_CTX_new()` and `BN_CTX_free()` functions are available in all versions on SSLeay and OpenSSL. The `BN_CTX_init()` function was added in SSLeay 0.9.1b.

SEE ALSO

Functions: *bn*, *err*, *BN_add*, *BN_CTX_start*

BN_CTX_start

NAME

BN_CTX_start, BN_CTX_get, BN_CTX_end – Use temporary BIGNUM variables

SYNOPSIS

```
#include <openssl/bn.h>
void BN_CTX_start(
       BN_CTX *ctx
);
BIGNUM *BN_CTX_get(
       BN_CTX *ctx
);
void BN_CTX_end(
       BN_CTX *ctx
);
```

DESCRIPTION

These functions are used to obtain temporary `BIGNUM` variables from a `BN_CTX` (which can been created by using the `BN_CTX_new()` function) in order to save the overhead of repeatedly creating and freeing `BIGNUM`s in functions that are called from inside a loop.

A function must call `BN_CTX_start()` first. Then, `BN_CTX_get()` may be called repeatedly to obtain temporary `BIGNUM`s. All `BN_CTX_get()` calls must be made before calling any other functions that use the `ctx` as an argument.

Finally, `BN_CTX_end()` must be called before returning from the function. When `BN_CTX_end()` is called, the `BIGNUM` pointers obtained from `BN_CTX_get()` become invalid.

RETURN VALUES

The `BN_CTX_start()` and `BN_CTX_end()` functions return no values.

The `BN_CTX_get()` function returns a pointer to the `BIGNUM`, or `NULL` on error. Once `BN_CTX_get()` has failed, the subsequent calls will return `NULL` as well, so it is sufficient to check the return value of the last `BN_CTX_get()` call. In case of an error, an error code is set, which can be obtained by `ERR_get_error()`.

HISTORY

The `BN_CTX_start()`, `BN_CTX_get()`, and `BN_CTX_end()` functions were added in OpenSSL 0.9.5.

SEE ALSO

Functions: *BN_CTX_new*

BN_generate_prime

NAME

BN_generate_prime, BN_is_prime, BN_is_prime_fasttest – Generate primes and test for primality

SYNOPSIS

```
#include <openssl/bn.h>

BIGNUM *BN_generate_prime(
        BIGNUM *ret, int num, int safe, BIGNUM *add, BIGNUM *rem, void (*callback)(int,
        int, void *), void *cb_arg
);

int BN_is_prime(
        const BIGNUM *a, int checks, void (*callback)(int, int, void *), BN_CTX *ctx,
        void *cb_arg
);

int BN_is_prime_fasttest(
        const BIGNUM *a, int checks, void (*callback)(int, int, void *), BN_CTX *ctx,
        void *cb_arg, int do_trial_division
);
```

DESCRIPTION

The `BN_generate_prime()` function generates a pseudo-random prime number of `num` bits. If `ret` is not `NULL`, it will be used to store the number.

If `callback` is not `NULL`, it is called as follows:

- `callback(0, i, cb_arg)` is called after generating the i-th potential prime number.
- While the number is being tested for primality, `callback(1, j, cb_arg)` is called as described below.
- When a prime has been found, `callback(2, i, cb_arg)` is called.

The prime may have to fulfill additional requirements for use in Diffie-Hellman key exchange:

If `add` is not `NULL`, the prime will fulfill the condition p % `add` == `rem` (p % `add` == 1 if `rem` == `NULL`) in order to suit a given generator.

If `safe` is true, it will be a safe prime (i.e. a prime p so that (p-1)/2 is also prime).

The PRNG must be seeded prior to calling `BN_generate_prime()`. The prime number generation has a negligible error probability.

`BN_is_prime()` and `BN_is_prime_fasttest()` test if the number `a` is prime. The following tests are performed until one of them shows that `a` is composite; if `a` passes all these tests, it is considered prime.

`BN_is_prime_fasttest()`, when called with `do_trial_division == 1`, first attempts trial division by a number of small primes; if no divisors are found by this test and `callback` is not `NULL`, `callback(1, -1, cb_arg)` is called. If `do_trial_division == 0`, this test is skipped.

Both `BN_is_prime()` and `BN_is_prime_fasttest()` perform a Miller-Rabin probabilistic primality test with `checks` iterations. If `checks == BN_prime_check`, a number of iterations is used that yields a false positive rate of at most 2^-80 for random input.

If callback is not NULL, callback(1, j, cb_arg) is called after the j-th iteration (j = 0, 1, ...). ctx is a pre-allocated BN_CTX (to save the overhead of allocating and freeing the structure in a loop), or NULL.

RETURN VALUES

The BN_generate_prime() function returns the prime number on success, NULL otherwise.

The BN_is_prime() function returns 0 if the number is composite, 1 if it is prime with an error probability of less than 0.25^checks, and -1 on error.

The error codes can be obtained by using the ERR_get_error() function.

HISTORY

The cb_arg arguments to BN_generate_prime() and to BN_is_prime() were added in SSLeay 0.9.0. The ret argument to BN_generate_prime() was added in SSLeay 0.9.1. The BN_is_prime_fasttest() function was added in OpenSSL 0.9.5.

SEE ALSO

Functions: *bn*, *err*, *rand*

bn_internal

NAME

bn_internal: bn_mul_words, bn_mul_add_words, bn_sqr_words, bn_div_wordsbn_add_words, bn_sub_words, bn_mul_comba4, bn_mul_comba8, bn_sqr_comba4, bn_sqr_comba8, bn_cmp_words, bn_mul_normal, bn_mul_low_normal, bn_mul_recursive, bn_mul_part_recursive, bn_mul_low_recursive, bn_mul_high, bn_sqr_normal, bn_sqr_recursive, bn_expand, bn_wexpand, bn_expand2, bn_fix_top, bn_check_top, bn_print, bn_dump, bn_set_max, bn_set_high, bn_set_low – BIGNUM library internal functions

SYNOPSIS

```
BN_ULONG bn_mul_words(
        BN_ULONG *rp, BN_ULONG *ap, int num, BN_ULONG w
);
BN_ULONG bn_mul_add_words(
         BN_ULONG *rp, BN_ULONG *ap, int num, BN_ULONG w
);
void bn_sqr_words(
        BN_ULONG *rp, BN_ULONG *ap, int num
);
BN_ULONG bn_div_words(
        BN_ULONG h, BN_ULONG l, BN_ULONG d
);
BN_ULONG bn_add_words(
        BN_ULONG *rp, BN_ULONG *ap, BN_ULONG *bp, int num
);
BN_ULONG bn_sub_words(
        BN_ULONG *rp, BN_ULONG *ap, BN_ULONG *bp, int num
);
void bn_mul_comba4(
        BN_ULONG *r, BN_ULONG *a, BN_ULONG *b
);
void bn_mul_comba8(
        BN_ULONG *r, BN_ULONG *a, BN_ULONG *b
);
void bn_sqr_comba4(
        BN_ULONG *r, BN_ULONG *a
);
void bn_sqr_comba8(
        BN_ULONG *r, BN_ULONG *a
);
```

```
int bn_cmp_words(
        BN_ULONG *a, BN_ULONG *b, int n
);
void bn_mul_normal(
        BN_ULONG *r, BN_ULONG *a, int na, BN_ULONG *b, int nb
);
void bn_mul_low_normal(
        BN_ULONG *r, BN_ULONG *a, BN_ULONG *b, int n
);
void bn_mul_recursive(
        BN_ULONG *r, BN_ULONG *a, BN_ULONG *b, int n2, BN_ULONG *tmp
);
void bn_mul_part_recursive(
         BN_ULONG *r, BN_ULONG *a, BN_ULONG *b, int tn, int n, BN_ULONG *tmp
);
void bn_mul_low_recursive(
         BN_ULONG *r, BN_ULONG *a, BN_ULONG *b, int n2, BN_ULONG *tmp
);
void bn_mul_high(
        BN_ULONG *r, BN_ULONG *a, BN_ULONG *b, BN_ULONG *l, int n2, BN_ULONG *tmp
);
void bn_sqr_normal(
        BN_ULONG *r, BN_ULONG *a, int n, BN_ULONG *tmp
);
void bn_sqr_recursive(
        BN_ULONG *r, BN_ULONG *a, int n2, BN_ULONG *tmp
);
void mul(
        BN_ULONG r, BN_ULONG a, BN_ULONG w, BN_ULONG c
);
void mul_add(
        BN_ULONG r, BN_ULONG a, BN_ULONG w, BN_ULONG c
);
void sqr(
        BN_ULONG r0, BN_ULONG r1, BN_ULONG a
);
BIGNUM *bn_expand(
        BIGNUM *a, int bits
);
BIGNUM *bn_wexpand(
```

```
        BIGNUM *a, int n
);
BIGNUM *bn_expand2(
        BIGNUM *a, int n
);
void bn_fix_top(
        BIGNUM *a
);
void bn_check_top(
        BIGNUM *a
);
void bn_print(
        BIGNUM *a
);
void bn_dump(
        BN_ULONG *d, int n
);
void bn_set_max(
        BIGNUM *a
);
void bn_set_high(
        BIGNUM *r, BIGNUM *a, int n
);
void bn_set_low(
        BIGNUM *r, BIGNUM *a, int n
);
```

DESCRIPTION

This page describes the internal functions used by the OpenSSL `BIGNUM` implementation. They are described here to facilitate debugging and extending the library. They are not to be used by applications.

The BIGNUM structure

```
typedef struct bignum_st
       {
       int top;      /* index of last used d (most significant word) */
       BN_ULONG *d;  /* pointer to an array of 'BITS2' bit chunks */
       int max;      /* size of the d array */
       int neg;      /* sign */
       } BIGNUM;
```

The big number is stored in `d`, a malloc array of `BN_ULONG`s, least significant first. A `BN_ULONG` can be either 16, 32 or 64 bits in size (`BITS2`), depending on the number of bits specified in `openssl/bn.h`.

The max is the size of the d array that has been allocated. The top is the last entry being used. For a value of 4, for example, bn.d[0]=4 and bn.top=1. The neg is 1 if the number is negative. When a BIGNUM is 0, the d field can be NULL and top == 0.

Various routines in this library require the use of temporary BIGNUM variables during their execution. Since dynamic memory allocation to create BIGNUMs is rather expensive when used in conjunction with repeated subroutine calls, the BN_CTX structure is used. This structure contains BN_CTX_NUM BIGNUMs. See *BN_CTX_start*.

Low-level arithmetic operations

These functions are implemented in C and for several platforms in assembly language:

`bn_mul_words(rp, ap, num, w)`

Operates on the num word arrays rp and ap. It computes ap * w, places the result in rp, and returns the high word (carry).

`bn_mul_add_words(rp, ap, num, w)`

Operates on the num word arrays rp and ap. It computes ap * w + rp, places the result in rp, and returns the high word (carry).

`bn_sqr_words(rp, ap, n)`

Operates on the num word array ap and the 2*num word array ap. It computes ap * ap word-wise, and places the low and high bytes of the result in rp.

`bn_div_words(h, l, d)`

Divides the two word number (h,l) by d and returns the result.

`bn_add_words(rp, ap, bp, num)`

Operates on the num word arrays ap, bp and rp. It computes ap + bp, places the result in rp, and returns the high word (carry).

`bn_sub_words(rp, ap, bp, num)`

Operates on the num word arrays ap, bp and rp. It computes ap - bp, places the result in rp, and returns the carry (1 if bp > ap, 0 otherwise).

`bn_mul_comba4(r, a, b)`

Operates on the 4 word arrays a and b and the 8 word array r. It computes a*b and places the result in r.

`bn_mul_comba8(r, a, b)`

Operates on the 8-word arrays a and b and the 16-word array r. It computes a*b and places the result in r.

`bn_sqr_comba4(r, a, b)`

Operates on the 4-word arrays a and b and the 8-word array r.

`bn_sqr_comba8(r, a, b)`

Operates on the 8-word arrays a and b and the 16-word array r.

The following functions are implemented in C:

`bn_cmp_words(a, b, n)`

Operates on the n word arrays a and b. It returns 1, 0 and -1 if a is greater than, equal and less than b.

bn_mul_normal(r, a, na, b, nb)

Operates on the na word array a, the nb word array b and the na+nb word array r. It computes a*b and places the result in r.

bn_mul_low_normal(r, a, b, n)

Operates on the n word arrays r, a and b. It computes the n low words of a*b and places the result in r.

bn_mul_recursive(r, a, b, n2, t)

Operates on the n2 word arrays a and b and the 2*n2 word arrays r and t. The n2 must be a power of 2. It computes a*b and places the result in r.

bn_mul_part_recursive(r, a, b, tn, n, tmp)

Operates on the n+tn word arrays a and b and the 4*n word arrays r and tmp.

bn_mul_low_recursive(r, a, b, n2, tmp)

Operates on the n2 word arrays r and tmp and the n2/2 word arrays a and b.

bn_mul_high(r, a, b, l, n2, tmp)

Operates on the n2 word arrays r, a, b and l (?) and the 3*n2 word array tmp.

BN_mul() calls bn_mul_normal(), or an optimized implementation if the factors have the same size: bn_mul_comba8() is used if they are 8 words long, bn_mul_recursive() if they are larger than BN_MULL_SIZE_NORMAL and the size is an exact multiple of the word size, and bn_mul_part_recursive() for others that are larger than BN_MULL_SIZE_NORMAL.

bn_sqr_normal(r, a, n, tmp)

Operates on the n word array a and the 2*n word arrays tmp and r.

The implementations use the following macros which, depending on the architecture, may use "long long" C operations or inline assembler. They are defined in bn_lcl.h.

mul(r, a, w, c)

Computes w*a+c and places the low word of the result in r and the high word in c.

mul_add(r, a, w, c)

Computes w*a+r+c and places the low word of the result in r and the high word in c.

sqr(r0, r1, a)

Computes a*a and places the low word of the result in r0 and the high word in r1.

Size changes

The bn_expand() macro ensures that b has enough space for a bits bit number. The bn_wexpand() macro ensures that b has enough space for an n word number. If the number has to be expanded, both macros call bn_expand2(), which allocates a new d array and copies the data. They return NULL on error, b otherwise.

The bn_fix_top() macro reduces a->top to point to the most significant non-zero word when a has shrunk.

Debugging

The `bn_check_top()` verifies that `((a)->top >= 0 && (a)->top <= (a)->max)`. A violation will cause the program to abort.

The `bn_print()` prints `a` to stderr. bn_dump() prints `n` words at `d` (in reverse order, i.e. most significant word first) to stderr.

The `bn_set_max()` makes `a` a static number with a `max` of its current size. This is used by `bn_set_low()` and `bn_set_high()` to make `r` a read-only `BIGNUM` that contains the `n` low or high words of `a`.

If `BN_DEBUG` is not defined, `bn_check_top()`, `bn_print()`, `bn_dump()`, and `bn_set_max()` are defined as empty macros.

SEE ALSO

Functions: *bn*

BN_mod_inverse

NAME

BN_mod_inverse – Compute inverse modulo n

SYNOPSIS

```
#include <openssl/bn.h>
BIGNUM *BN_mod_inverse(
       BIGNUM *r, BIGNUM *a, const BIGNUM *n, BN_CTX *ctx
);
```

DESCRIPTION

The `BN_mod_inverse()` function computes the inverse of `a` modulo `n` and places the result in `r` (`(a*r)%n==1`). If `r` is NULL, a new `BIGNUM` is created.

The `ctx` is a previously allocated `BN_CTX` used for temporary variables. The `r` value may be the same `BIGNUM` as `a` or `n`.

RETURN VALUES

The `BN_mod_inverse()` function returns the `BIGNUM` containing the inverse, and NULL on error. The error codes can be obtained by using `ERR_get_error()`.

HISTORY

The `BN_mod_inverse()` function is available in all versions of SSLeay and OpenSSL.

SEE ALSO

Functions: *bn*, *err*, *BN_add*

BN_mod_mul_montgomery

NAME

BN_mod_mul_montgomery , BN_MONT_CTX_new, BN_MONT_CTX_init, BN_MONT_CTX_free, BN_MONT_CTX_set, BN_MONT_CTX_copy, BN_from_montgomery, BN_to_montgomery – Montgomery multiplication

SYNOPSIS

```
#include <openssl/bn.h>

BN_MONT_CTX *BN_MONT_CTX_new(
        void
);

void BN_MONT_CTX_init(
       BN_MONT_CTX *ctx
);

void BN_MONT_CTX_free(
       BN_MONT_CTX *mont
);

 int BN_MONT_CTX_set(
       BN_MONT_CTX *mont, const BIGNUM *m, BN_CTX *ctx
);

BN_MONT_CTX *BN_MONT_CTX_copy(
        BN_MONT_CTX *to, BN_MONT_CTX *from
);

int BN_mod_mul_montgomery(
        BIGNUM *r, BIGNUM *a, BIGNUM *b, BN_MONT_CTX *mont, BN_CTX *ctx
);

int BN_from_montgomery(
       BIGNUM *r, BIGNUM *a, BN_MONT_CTX *mont, BN_CTX *ctx
);

int BN_to_montgomery(
       BIGNUM *r, BIGNUM *a, BN_MONT_CTX *mont, BN_CTX *ctx
);
```

DESCRIPTION

These functions implement Montgomery multiplication. They are used automatically when `BN_mod_exp()` is called with suitable input, but they may be useful when several operations are performed using the same modulus.

The `BN_MONT_CTX_new()` function allocates and initializes a `BN_MONT_CTX` structure. The `BN_MONT_CTX_init()` function initializes an existing uninitialized `BN_MONT_CTX`.

The BN_MONT_CTX_set() function sets up the mont structure from the modulus m by precomputing its inverse and a value R.

The BN_MONT_CTX_copy() function copies the BN_MONT_CTX from to to.

The BN_MONT_CTX_free() function frees the components of the BN_MONT_CTX, and, if it was created by BN_MONT_CTX_new(), also the structure itself.

The BN_mod_mul_montgomery() function computes Mont(a,b):=a*b*R^-1 and places the result in r.

The BN_from_montgomery() function performs the Montgomery reduction r = a*R^-1.

The BN_to_montgomery() function computes Mont(a,R^2), i.e. a*R.

For all functions, ctx is a previously allocated BN_CTX used for temporary variables.

The BN_MONT_CTX structure is defined as follows:

```
 typedef struct bn_mont_ctx_st
        {
        int ri;         /* number of bits in R */
        BIGNUM RR;      /* R^2 (used to convert to Montgomery form) */
        BIGNUM N;       /* The modulus */
        BIGNUM Ni;      /* R*(1/R mod N) - N*Ni = 1
                         * (Ni is only stored for bignum algorithm) */
        BN_ULONG n0;    /* least significant word of Ni */

        int flags;
        } BN_MONT_CTX;
```

BN_to_montgomery() is a macro.

RETURN VALUES

The BN_MONT_CTX_new() function returns the newly allocated BN_MONT_CTX, and NULL on error.

The BN_MONT_CTX_init() and BN_MONT_CTX_free() functions have no return values.

For the other functions, 1 is returned for success, 0 on error. The error codes can be obtained by using ERR_get_error().

HISTORY

The BN_MONT_CTX_new(), BN_MONT_CTX_free(), BN_MONT_CTX_set(), BN_mod_mul_montgomery(), BN_from_montgomery(), and BN_to_montgomery() functions are available in all versions of SSLeay and OpenSSL.

The BN_MONT_CTX_init() and BN_MONT_CTX_copy() functions were added in SSLeay 0.9.1b.

SEE ALSO

Functions: *bn*, *err*, *BN_add*, *BN_CTX_new*

BN_mod_mul_reciprocal

NAME

BN_mod_mul_reciprocal, BN_div_recp, BN_RECP_CTX_new, BN_RECP_CTX_init, BN_RECP_CTX_free, BN_RECP_CTX_set – Modular multiplication using reciprocal

SYNOPSIS

```
#include <openssl/bn.h>

BN_RECP_CTX *BN_RECP_CTX_new(
        void
);

void BN_RECP_CTX_init(
       BN_RECP_CTX *recp
);

void BN_RECP_CTX_free(
       BN_RECP_CTX *recp
);

int BN_RECP_CTX_set(
       BN_RECP_CTX *recp, const BIGNUM *m, BN_CTX *ctx
);

int BN_div_recp(
       BIGNUM *dv, BIGNUM *rem, BIGNUM *a, BN_RECP_CTX *recp, BN_CTX *ctx
);

int BN_mod_mul_reciprocal(
        BIGNUM *r, BIGNUM *a, BIGNUM *b, BN_RECP_CTX *recp, BN_CTX *ctx
);
```

DESCRIPTION

The `BN_mod_mul_reciprocal()` function can be used to perform an efficient `BN_mod_mul()` operation when the operation will be performed repeatedly with the same modulus. It computes `r=(a*b)%m` using `recp=1/m`, which is set as described below. `ctx` is a previously allocated `BN_CTX` used for temporary variables.

The `BN_RECP_CTX_new()` function allocates and initializes a `BN_RECP` structure. The `BN_RECP_CTX_init()` function initializes an existing uninitialized `BN_RECP`.

The `BN_RECP_CTX_free()` function frees the components of the `BN_RECP`, and, if it was created by `BN_RECP_CTX_new()`, also the structure itself.

The `BN_RECP_CTX_set()` function stores `m` in `recp` and sets it up for computing 1/`m` and shifting it left by `BN_num_bits(m)`+1 to make it an integer. The result and the number of bits it was shifted left will later be stored in `recp`.

The `BN_div_recp()` function divides `a` by `m` using `recp`. It places the quotient in `dv` and the remainder in `rem`.

The `BN_RECP_CTX` structure is defined as follows:

```
 typedef struct bn_recp_ctx_st
{
BIGNUM N;/* the divisor */
BIGNUM Nr;/* the reciprocal */
int num_bits;
int shift;
int flags;
} BN_RECP_CTX;
```

It cannot be shared between threads.

RETURN VALUES

The `BN_RECP_CTX_new()` function returns the newly allocated `BN_RECP_CTX`, and NULL on error.

The `BN_RECP_CTX_init()` and `BN_RECP_CTX_free()` functions have no return values.

For the other functions, 1 is returned for success, 0 on error. The error codes can be obtained by *ERR_get_error*.

HISTORY

The `BN_RECP_CTX` structure was added in SSLeay 0.9.0. Before that, the `BN_reciprocal()` function was used instead, and the `BN_mod_mul_reciprocal()` arguments were different.

SEE ALSO

Functions: *bn*, *err*, *BN_add*, *BN_CTX_new*

BN_new

NAME

BN_new, BN_init, BN_clear, BN_free, BN_clear_free – Allocate and free BIGNUMs

SYNOPSIS

```
#include <openssl/bn.h>
BIGNUM *BN_new(
        void
);
void BN_init(
        BIGNUM *
);
void BN_clear(
        BIGNUM *a
);
void BN_free(
        BIGNUM *a
);
void BN_clear_free(
        BIGNUM *a
);
```

DESCRIPTION

The `BN_new()` function allocates and initializes a `BIGNUM` structure. The `BN_init()` function initializes an existing uninitialized `BIGNUM`.

The `BN_clear()` function is used to destroy sensitive data such as keys when they are no longer needed. It erases the memory used by `a` and sets it to the value 0.

The `BN_free()` function frees the components of the `BIGNUM`, and if it was created by `BN_new()`, also the structure itself. The `BN_clear_free()` function overwrites the data before the memory is returned to the system.

RETURN VALUES

The `BN_new()` function returns a pointer to the `BIGNUM`. If the allocation fails, it returns `NULL` and sets an error code that can be obtained by using `ERR_get_error()`.

The `BN_init()`, `BN_clear()`, `BN_free()`, and `BN_clear_free()` functions have no return values.

HISTORY

The `BN_new()`, `BN_clear()`, `BN_free()`, and `BN_clear_free()` functions are available in all versions on SSLeay and OpenSSL. The `BN_init()` function was added in SSLeay 0.9.1b.

BN_num_bytes

NAME

BN_num_bytes, BN_num_bits, BN_num_bits_word – Get BIGNUM size

SYNOPSIS

```
#include <openssl/bn.h>
int BN_num_bytes(
       const BIGNUM *a
);
int BN_num_bits(
       const BIGNUM *a
);
int BN_num_bits_word(
       BN_ULONG w
);
```

DESCRIPTION

These functions return the size of a `BIGNUM` in bytes or bits, and the size of an unsigned integer in bits.

`BN_num_bytes()` is a macro.

RETURN VALUES

The size.

HISTORY

The `BN_num_bytes()`, `BN_num_bits()`, and `BN_num_bits_word()` are available in all versions of SSLeay and OpenSSL.

SEE ALSO

Functions: *bn*

BN_rand

NAME

BN_rand, BN_pseudo_rand – Generate pseudo-random number

SYNOPSIS

```
#include <openssl/bn.h>
int BN_rand(
        BIGNUM *rnd, int bits, int top, int bottom
);
int BN_pseudo_rand(
        BIGNUM *rnd, int bits, int top, int bottom
);
int BN_rand_range(
        BIGNUM *rnd, BIGNUM *range
);
```

DESCRIPTION

The `BN_rand()` function generates a cryptographically strong pseudo-random number of `bits` in length and stores it in `rnd`. If `top` is -1, the most significant bit of the random number can be zero. If `top` is 0, it is set to 1, and if `top` is 1, the two most significant bits of the number will be set to 1, so that the product of two such random numbers will always have 2*`bits` length. If `bottom` is true, the number will be odd.

The `BN_pseudo_rand()` function does the same, but pseudo-random numbers generated by this function are not necessarily unpredictable. They can be used for non-cryptographic purposes and for certain purposes in cryptographic protocols, but usually not for key generation etc.

The `BN_rand_range()` function generates a cryptographically strong pseudo-random number `rnd` in the range 0 <lt>= `rnd` < `range`.

The PRNG must be seeded prior to calling the `BN_rand()` or `BN_rand_range()` functions.

RETURN VALUES

The functions return 1 on success, 0 on error. The error codes can be obtained by using `ERR_get_error()`.

HISTORY

The `BN_rand()` function is available in all versions of SSLeay and OpenSSL. The `BN_pseudo_rand()` function was added in OpenSSL 0.9.5. The `top` == -1 case and the `BN_rand_range()` function were added in OpenSSL 0.9.6a.

SEE ALSO

Functions: *bn*, *err*, *rand*, *RAND_add*, *RAND_bytes*

BN_set_bit

NAME

BN_set_bit, BN_clear_bit, BN_is_bit_set, BN_mask_bits, BN_lshift, BN_lshift1 BN_rshift, BN_rshift1 – Bit operations on BIGNUMs

SYNOPSIS

```
#include <openssl/bn.h>
int BN_set_bit(
        BIGNUM *a, int n
);
int BN_clear_bit(
        BIGNUM *a, int n
);
int BN_is_bit_set(
        const BIGNUM *a, int n
);
int BN_mask_bits(
        BIGNUM *a, int n
);
int BN_lshift(
        BIGNUM *r, const BIGNUM *a, int n
);
int BN_lshift1(
        BIGNUM *r, BIGNUM *a
);
int BN_rshift(
        BIGNUM *r, BIGNUM *a, int n
);
int BN_rshift1(
        BIGNUM *r, BIGNUM *a
);
```

DESCRIPTION

The `BN_set_bit()` function sets bit `n` in `a` to 1 (`a|=(1<<n)`). The number is expanded if necessary.

The `BN_clear_bit()` function sets bit `n` in `a` to 0 (`a&=~(1<<n)`). An error occurs if `a` is shorter than `n` bits.

The `BN_is_bit_set()` function tests if bit `n` in `a` is set.

The `BN_mask_bits()` function truncates `a` to an `n` bit number (`a&=~((~0)>>n)`). An error occurs if `a` already is shorter than `n` bits.

The BN_lshift() function shifts a left by n bits and places the result in r (r=a*2^n). The BN_lshift1() function shifts a left by one and places the result in r (r=2*a).

The BN_rshift() function shifts a right by n bits and places the result in r (r=a/2^n). The BN_rshift1() function shifts a right by one and places the result in r (r=a/2).

For the shift functions, r and a may be the same variable.

RETURN VALUES

The BN_is_bit_set() function returns 1 if the bit is set, 0 otherwise.

All other functions return 1 for success, 0 on error. The error codes can be obtained by using ERR_get_error().

HISTORY

The BN_set_bit(), BN_clear_bit(), BN_is_bit_set(), BN_mask_bits(), BN_lshift(), BN_lshift1(), BN_rshift(), and BN_rshift1() functions are available in all versions of SSLeay and OpenSSL.

SEE ALSO

Functions: *bn*, *BN_num_bytes*, *BN_add*

BN_zero

NAME

BN_zero, BN_one, BN_value_one, BN_set_word, BN_get_word – BIGNUM assignment operations

SYNOPSIS

```
#include <openssl/bn.h>
int BN_zero(
        BIGNUM *a
);
int BN_one(
        BIGNUM *a
);
BIGNUM *BN_value_one(
        void
);
int BN_set_word(
        BIGNUM *a, unsigned long w
);
unsigned long BN_get_word(
         BIGNUM *a
);
```

DESCRIPTION

The `BN_zero()`, `BN_one()`, and `BN_set_word()` functions set `a` to the values 0, 1 and `w` respectively. `BN_zero()` and `BN_one()` are macros.

The `BN_value_one()` function returns a `BIGNUM` constant of value 1. This constant is useful for use in comparisons and assignment.

The `BN_get_word()` function returns `a`, if it can be represented as an unsigned long.

RETURN VALUES

The `BN_get_word()` function returns the value `a`, and 0xffffffffL if `a` cannot be represented as an unsigned long.

The `BN_zero()`, `BN_one()`, and `BN_set_word()` functions return 1 on success, 0 otherwise. The `BN_value_one()` function returns the constant.

RESTRICTIONS

Someone might change the constant.

If a `BIGNUM` is equal to 0xffffffffL it can be represented as an unsigned long but this value is also returned on error.

HISTORY

The `BN_zero()`, `BN_one()`, and `BN_set_word()` functions are available in all versions of SSLeay and OpenSSL. The `BN_value_one()` and `BN_get_word()` functions were added in SSLeay 0.8.

SEE ALSO

Functions: *bn*, *BN_bn2bin*

buffer

NAME

buffer: BUF_MEM_new, BUF_MEM_free, BUF_MEM_grow, BUF_strdup – Simple character arrays structure

SYNOPSIS

```
#include <openssl/buffer.h>

BUF_MEM *BUF_MEM_new(
        void
);

voidBUF_MEM_free(
        BUF_MEM *a
);

intBUF_MEM_grow(
        BUF_MEM *str, int len
);

char * BUF_strdup(
        const char *str
);
```

DESCRIPTION

The buffer library handles simple character arrays. Buffers are used for various purposes in the library, most notably memory BIOs.

The library uses the BUF_MEM structure defined in buffer.h:

```
typedef struct buf_mem_st
{
       int length;      /* current number of bytes */
       char *data;
       int max;         /* size of buffer */
} BUF_MEM;
```

The `length` is the current size of the buffer in bytes, `max` is the amount of memory allocated to the buffer. There are three functions which handle these and one miscellaneous function.

The `BUF_MEM_new()` function allocates a new buffer of zero size.

The `BUF_MEM_free()` function frees up an already existing buffer. The data is zeroed before freeing up in case the buffer contains sensitive data.

The `BUF_MEM_grow()` function changes the size of an already existing buffer to `len`. Any data already in the buffer is preserved if it increases in size.

The `BUF_strdup()` function copies a null terminated string into a block of allocated memory and returns a pointer to the allocated block. Unlike the standard C library `strdup()`, this function uses `OPENSSL_malloc()`. It should be used in preference to the standard library `strdup()` because it can be used for memory leak checking or replacing the `malloc()` function.

The memory allocated from the `BUF_strdup()` function should be freed up using the `OPENSSL_free()` function.

RETURN VALUES

The `BUF_MEM_new()` function returns the buffer or NULL on error.

The `BUF_MEM_free()` function has no return value.

The `BUF_MEM_grow()` function returns zero on error or the new size (i.e. `len`).

HISTORY

The `BUF_MEM_new()`, `BUF_MEM_free()`, and `BUF_MEM_grow()` functions are available in all versions of SSLeay and OpenSSL. The `BUF_strdup()` function was added in SSLeay 0.8.

SEE ALSO

Function: *bio*

ca

NAME

ca – Sample minimal CA application

SYNOPSIS

openssl ca [-verbose] [-config *filename*] [-name *section*] [-revoke *file*] [-gencrl] [-crldays *days*] [-crlhours *hours*] [-crlexts *section*] [-startdate *date*] [-enddate *date*] [-days *arg*] [-md *arg*] [-policy *arg*] [-keyfile *arg*] [-key *arg*] [-passin *arg*] [-cert *file*] [-in *file*] [-out *file*] [-notext] [-outdir *dir*] [-infiles] [-spkac *file*] [-ss_cert *file*] [-preserveDN] [-batch] [-msie_hack] [-extensions *section*]

CA OPTIONS

verbose

Prints extra details about the operations being performed.

config *filename*

Specifies the configuration file to use.

name *section*

Specifies the configuration file section to use. Overrides `default_ca` in the `ca` section.

in *filename*

An input filename containing a single certificate request to be signed by the CA.

ss_cert *filename*

A single self signed certificate to be signed by the CA.

spkac *filename*

A file containing a single Netscape signed public key and challenge and additional field values to be signed by the CA. See the NOTES section for information on the required format.

infiles

If present this should be the last option, all subsequent arguments are assumed to the the names of files containing certificate requests.

out *filename*

The output file to output certificates to. The default is standard output. The certificate details will also be printed out to this file.

outdir *directory*

The directory to output certificates to. The certificate will be written to a filename consisting of the serial number in hex with `.pem` appended.

cert

The CA certificate file.

keyfile *filename*

The private key to sign requests with.

`key` *password*

The password used to encrypt the private key. Since on some systems the command line arguments are visible (e.g. UNIX with the `ps` utility) this option should be used with caution.

`passin` *arg*

The key password source. For more information about the format of `arg` see the Pass Prhase Arguments section in *openssl*.

`notext`

Does not output the text form of a certificate to the output file.

`startdate` *date*

Allows the start date to be explicitly set. The format of the date is YYMMDDHHMMSSZ (the same as an ASN1 UTCTime structure).

`enddate` *date*

Allows the expiration date to be explicitly set. The format of the date is YYMMDDHHMMSSZ (the same as an ASN1 UTCTime structure).

`days` *arg*

The number of days to certify the certificate for.

`md` *arg*

The message digest to use. Possible values include md5, sha1 and mdc2. This option also applies to CRLs.

`policy` *arg*

Defines the CA policy to use. This is a section in the configuration file which decides which fields should be mandatory or match the CA certificate. See the `POLICY FORMAT` section for more information.

`msie_hack`

A legacy option to make `ca` work with very old versions of the IE certificate enrollment control `certenr3`. It used UniversalStrings for almost everything. Since the old control has various security bugs its use is strongly discouraged. The newer control `Xenroll` does not need this option.

`preserveDN`

Normally the DN order of a certificate is the same as the order of the fields in the relevant policy section. When this option is set the order is the same as the request. This is largely for compatibility with the older IE enrollment control which would only accept certificates if their DNs match the order of the request. This is not needed for Xenroll.

`batch`

Sets the batch mode. In this mode no questions will be asked and all certificates will be certified automatically.

`extensions` *section*

The section of the configuration file containing certificate extensions to be added when a certificate is issued. If no extension section is present then a V1 certificate is created. If the extension section is present (even if it is empty) then a V3 certificate is created.

CRL OPTIONS

`gencrl`

Generates a CRL based on information in the index file.

`crldays` *num*

The number of days before the next CRL is due. That is the days from now to place in the CRL next Update field.

`crlhours` *num*

The number of hours before the next CRL is due.

`revoke` *filename*

A filename containing a certificate to revoke.

`crlexts` *section*

The section of the configuration file containing CRL extensions to include. If no CRL extension section is present then a V1 CRL is created, if the CRL extension section is present (even if it is empty) then a V2 CRL is created. The CRL extensions specified are CRL extensions and `not` CRL entry extensions. It should be noted that some software (such as Netscape) cannot handle V2 CRLs.

CONFIGURATION FILE OPTIONS

The section of the configuration file containing options for `ca` is found as follows:

If the `name` command line option is used, then it names the section to be used. Otherwise, the section to be used must be named in the `default_ca` option of the `ca` section of the configuration file (or in the default section of the configuration file). Besides `default_ca`, the following options are read directly from the `ca` section:

RANDFILE
preserve
msie_hack

With the exception of RANDFILE, this is probably a bug and may change in future releases.

Many of the configuration file options are identical to command line options. Where the option is present in the configuration file and the command line, the command line value is used. Where an option is described as mandatory, then it must be present in the configuration file or the command line equivalent (if any) is used.

`oid_file`

Specifies a file containing additional `OBJECT IDENTIFIERS`. Each line of the file should consist of the numerical form of the object identifier followed by white space then the short name followed by white space and finally the long name.

`oid_section`

Specifies a section in the configuration file containing extra object identifiers. Each line should consist of the short name of the object identifier followed by = and the numerical form. The short and long names are the same when this option is used.

`new_certs_dir`

The same as the `outdir` command line option. It specifies the directory where new certificates will be placed. Mandatory.

`certificate`

The same as the `cert` option. It gives the file containing the CA certificate. Mandatory.

`private_key`

The same as the `keyfile` option. The file containing the CA private key. Mandatory.

`RANDFILE`

A file used to read and write random number seed information, or an EGD socket (see *RAND_egd*).

`default_days`

The same as the `days` option. The number of days to certify a certificate for.

`default_startdate`

The same as the `startdate` option. The start date to certify a certificate for. If not set the current time is used.

`default_enddate`

The same as the `enddate` option. Either this option or `default_days` (or the command line equivalents) must be present.

`default_crl_hours default_crl_days`

The same as the `crlhours` and the `crldays` options. These will only be used if neither command line option is present. At least one of these must be present to generate a CRL.

`default_md`

The same as the `-md` option. The message digest to use. Mandatory.

`database`

The text database file to use. Mandatory. This file must be present though initially it will be empty.

`serialfile`

A text file containing the next serial number to use in hex. Mandatory. This file must be present and contain a valid serial number.

`x509_extensions`

The same as the `extensions` option.

`crl_extensions`

The same as the `crlexts` option.

`preserve`

The same as the `preserveDN` option.

`msie_hack`

The same as the `msie_hack` option.

`policy`

The same as the `policy` option. Mandatory.

POLICY FORMAT

The policy section consists of a set of variables corresponding to certificate DN fields. If the value is `match` then the field value must match the same field in the CA certificate. If the value is `supplied` then it must be present. If the value is `optional` then it may be present. Any fields not mentioned in the policy section are silently deleted, unless the `preserveDN` option is set but this can be regarded more of a quirk than intended behavior.

SPKAC FORMAT

The input to the `spkac` command line option is a Netscape signed public key and challenge. This will usually come from the `KEYGEN` tag in an HTML form to create a new private key. It is however possible to create SPKACs using the `spkac` utility.

The file should contain the variable SPKAC set to the value of the SPKAC and also the required DN components as name value pairs. If you need to include the same component twice then it can be preceded by a number and a '.'.

DESCRIPTION

The `ca` command is a minimal CA application. It can be used to sign certificate requests in a variety of forms and generate CRLs it also maintains a text database of issued certificates and their status.

The options descriptions will be divided into each purpose.

NOTES

The `ca` utility originally was meant as an example of how to do things in a CA. It was not supposed be be used as a full blown CA; nevertheless, some people are using it for this purpose.

The `ca` command is effectively a single user command. No locking is done on the various files and attempts to run more than one `ca` command on the same database can have unpredictable results.

RESTRICTIONS

The text database index file is a critical part of the process and if corrupted it can be difficult to fix. It is theoretically possible to rebuild the index file from all the issued certificates and a current CRL. However, there is no option to do this.

CRL entry extensions cannot currently be created. Only CRL extensions can be added.

V2 CRL features like delta CRL support and CRL numbers are not currently supported.

Although several requests can be input and handled at once it is only possible to include one SPKAC or self signed certificate.

The use of an in memory text database can cause problems when large numbers of certificates are present because, as the name implies the database has to be kept in memory.

Certificate request extensions are ignored. Some kind of policy should be included to use certain static extensions and certain extensions from the request.

It is not possible to certify two certificates with the same DN. This is a side effect of how the text database is indexed and it cannot easily be fixed without introducing other problems. Some S/MIME clients can use two certificates with the same DN for separate signing and encryption keys.

The ca command really needs rewriting or the required functionality exposed at either a command or interface level so a more friendly utility (perl script or GUI) can handle things properly. The scripts CA.sh and CA.pl help a little but not very much.

Any fields in a request that are not present in a policy are silently deleted. This does not happen if the preserveDN option is used but the extra fields are not displayed when the user is asked to certify a request. The behaviour should be more friendly and configurable.

Cancelling some commands by refusing to certify a certificate can create an empty file.

EXAMPLES

These examples assume that the ca directory structure is already set up and the relevant files already exist. This usually involves creating a CA certificate and private key with req, a serial number file and an empty index file and placing them in the relevant directories.

To use the sample configuration file below the directories demoCA, demoCA/private and demoCA/newcerts would be created. The CA certificate would be copied to demoCA/cacert.pem and its private key to demoCA/private/cakey.pem . A file demoCA/serial would be created containing for example "01" and the empty index file demoCA/index.txt.

Sign a certificate request:

```
openssl ca -in req.pem -out newcert.pem
```

Sign a certificate request, using CA extensions:

```
openssl ca -in req.pem -extensions v3_ca -out newcert.pem
```

Generate a CRL

```
openssl ca -gencrl -out crl.pem
```

Sign several requests:

```
openssl ca -infiles req1.pem req2.pem req3.pem
```

Certify a Netscape SPKAC:

```
openssl ca -spkac spkac.txt
```

A sample SPKAC file (the SPKAC line has been truncated for clarity):

```
SPKAC=MIG0MGAwXDANBgkqhkiG9w0BAQEFAANLADBIAkEAn7PDhCeV/xIxUg8V70YRxK2A5
CN=Steve Test
emailAddress=steve@openssl.org
0.OU=OpenSSL Group
1.OU=Another Group
```

A sample configuration file with the relevant sections for ca:

```
[ ca ]

default_ca      = CA_default            # The default ca section

[ CA_default ]

dir            = ./demoCA              # top dir
database       = $dir/index.txt        # index file.
new_certs_dir= $dir/newcerts         # new certs dir
certificate    = $dir/cacert.pem       # The CA cert
serial         = $dir/serial           # serial no file
```

```
private_key    = $dir/private/cakey.pem# CA private key
RANDFILE       = $dir/private/.rand    # random number file

default_days   = 365                   # how long to certify for
default_crl_days= 30                   # how long before next CRL
default_md     = md5                   # md to use
policy         = policy_any            # default policy

[ policy_any ]

countryName            = supplied
stateOrProvinceName    = optional
organizationName       = optional
organizationalUnitName = optional
commonName             = supplied
emailAddress           = optional
```

ENVIRONMENT VARIABLES

`OPENSSL_CONF` reflects the location of master configuration file it can be overridden by the `config` command line option.

FILES

Note: the location of all files can change either by compile time options, configuration file entries, environment variables or command line options. The values below reflect the default values.

```
/usr/local/ssl/lib/openssl.cnf - master configuration file
./demoCA                       - main CA directory
./demoCA/cacert.pem            - CA certificate
./demoCA/private/cakey.pem     - CA private key
./demoCA/serial                - CA serial number file
./demoCA/serial.old            - CA serial number backup file
./demoCA/index.txt             - CA text database file
./demoCA/index.txt.old         - CA text database backup file
./demoCA/certs                 - certificate output file
./demoCA/.rnd                  - CA random seed information
```

SEE ALSO

Commands: *req*, *spkac*, *x509*, *CA.pl*

Others: *config*

ca.pl

NAME

ca.pl – Friendlier interface for OpenSSL certificate programs

SYNOPSIS

```
openssl ca.pl [-?] [-h] [-help] [-newcert] [-newreq] [-newca] [-pkcs12] [-sign]
[-signreq] [-xsign] [-signCA] [-signcert] [-verify] [-files]
```

CA OPTIONS

`?, h, help`

Prints a usage message.

newcert

The private key and certificate are written to the file `newreq.pem`.

newreq

Creates a new certificate request. The private key and request are written to the file `newreq.pem`.

newca

Creates a new CA hierarchy for use with the `ca` program (or the `signcert` and `xsign` options). The user is prompted to enter the filename of the CA certificates (which should also contain the private key) or, by hitting ENTER details of the CA will be prompted for. The relevant files and directories are created in a directory called `demoCA` in the current directory.

pkcs12

Creates a PKCS#12 file containing the user certificate, private key and CA certificate. It expects the user certificate and private key to be in the file `newcert.pem` and the CA certificate to be in the file `demoCA/cacert.pem`, it creates a file `newcert.p12`. This command can thus be called after the `sign` option. The PKCS#12 file can be imported directly into a browser. If there is an additional argument on the command line it will be used as the `friendly name` for the certificate (which is typically displayed in the browser list box), otherwise the name `My Certificate` is used.

sign, signreq, xsign

Calls the `ca` program to sign a certificate request. It expects the request to be in the file `newreq.pem`. The new certificate is written to the file `newcert.pem` except in the case of the `xsign` option when it is written to standard output.

signCA

The same as the `signreq` option except it uses the configuration file section `v3_ca` and so makes the signed request a valid CA certificate. This is useful when creating intermediate CA from a root CA.

signcert

The same as `sign` option, except it expects a self signed certificate to be present in the file `newreq.pem`.

verify

Verifies certificates against the CA certificate for demoCA. If no certificates are specified on the command line it tries to verify the file newcert.pem.

files

One or more optional certificate file names for use with the verify option.

DESCRIPTION

The ca.pl script is a perl script that supplies the relevant command line arguments to the openssl command for some common certificate operations. It is intended to simplify the process of certificate creation and management by the use of some simple options.

NOTES

Most of the filenames mentioned can be modified by editing the CA.pl script.

If the demoCA directory already exists then the newca option will not overwrite it and will do nothing. This can happen if a previous call using the newca option terminated abnormally. To get the correct behavior, delete the demoCA directory if it already exists.

Under some environments it may not be possible to run the CA.pl script directly (for example Win32), and the default configuration file location may be wrong. In this case the command perl -S CA.pl can be used and the OPENSSL_CONF environment variable changed to point to the correct path of the configuration file openssl.cnf.

The script is intended as a simple front end for the openssl program for use by a beginner. Its behavior isn't always what is wanted. For more control over the behavior of the certificate commands call the openssl command directly.

EXAMPLES

Create a CA hierarchy:

```
CA.pl -newca
```

Complete certificate creation example: create a CA, create a request, sign the request and finally create a PKCS#12 file containing it:

```
CA.pl -newca
CA.pl -newreq
CA.pl -signreq
CA.pl -pkcs12 "My Test Certificate"
```

DSA CERTIFICATES

Although the CA.pl creates RSA CAs and requests, it is still possible to use it with DSA certificates and requests using the req command directly. The following example shows the steps that would typically be taken.

Create some DSA parameters:

```
openssl dsaparam -out dsap.pem 1024
```

Create a DSA CA certificate and private key:

```
openssl req -x509 -newkey dsa:dsap.pem -keyout cacert.pem -out cacert.pem
```

Create the CA directories and files:

```
CA.pl -newca
```

Enter `cacert.pem` when prompted for the CA file name.

Create a DSA certificate request and private key (a different set of parameters can optionally be created first):

```
openssl req -out newreq.pem -newkey dsa:dsap.pem
```

Sign the request:

```
CA.pl -signreq
```

ENVIRONMENT VARIABLES

The variable `OPENSSL_CONF` if defined allows an alternative configuration file location to be specified. It should contain the full path to the configuration file, not just its directory.

SEE ALSO

Commands: *x509*, *ca*, *req, plcs12*

Others: *config*

ciphers

NAME

ciphers – SSL cipher display and cipher list tool

SYNOPSIS

```
openssl ciphers [-ssl2] [-ssl3] [-tls1] [cipherlist]
```

OPTIONS

`v`

Verbose option. Lists ciphers with a complete description of protocol version (SSLv2 or SSLv3; the latter includes TLS), key exchange, authentication, encryption and mac algorithms used along with any key size restrictions and whether the algorithm is classed as an export cipher. Without the `v` option, ciphers may seem to appear twice in a cipher list; this is when similar ciphers are available for SSL v2 and for SSL v3/TLS v1.

`ssl3`

Only include SSL v3 ciphers.

`ssl2`

Only include SSL v2 ciphers.

`tls1`

Only include TLS v1 ciphers.

`h`, `?`

Prints a brief usage message.

`cipherlist`

A cipher list to convert to a cipher preference list. If it is not included then the default cipher list will be used.

CIPHER LIST FORMAT

The cipher list consists of one or more *`cipher strings`* separated by colons. Commas or spaces are also acceptable separators but colons are normally used.

The actual cipher string can take several different forms.

It can consist of a single cipher suite such as `RC4-SHA`.

It can represent a list of cipher suites containing a certain algorithm, or cipher suites of a certain type. For example `SHA1` represents all ciphers suites using the digest algorithm SHA1, and `SSLv3` represents all SSL v3 algorithms.

Lists of cipher suites can be combined in a single cipher string using the `+` character. This is used as a logical `and` operation. For example `SHA1+DES` represents all cipher suites containing the SHA1 `and` the DES algorithms.

Each cipher string can be optionally preceded by the characters `!`, `-` or `+`.

If `!` is used then the ciphers are permanently deleted from the list. The ciphers deleted can never reappear in the list even if they are explicitly stated.

If – is used then the ciphers are deleted from the list, but some or all of the ciphers can be added again by later options.

If + is used then the ciphers are moved to the end of the list. This option doesn't add any new ciphers it just moves matching existing ones.

If none of these characters is present then the string is just interpreted as a list of ciphers to be appended to the current preference list. If the list includes any ciphers already present they will be ignored; that is, they will not move to the end of the list.

Additionally the cipher string `@STRENGTH` can be used at any point to sort the current cipher list in order of encryption algorithm key length.

CIPHER STRINGS

Following is a list of all permitted cipher strings and their meanings.

`DEFAULT`

The default cipher list. This is determined at compile time and is normally `ALL:!ADH:RC4+RSA:+SSLv2:@STRENGTH`. This must be the first cipher string specified.

`ALL`

All cipher suites except the `eNULL` ciphers which must be explicitly enabled.

`HIGH`

Hiigh encryption cipher suites. This currently means those with key lengths larger than 128 bits.

`MEDIUM`

Medium encryption cipher suites, currently those using 128 bit encryption.

`LOW`

Low encryption cipher suites, currently those using 64 or 56 bit encryption algorithms but excluding export cipher suites.

`EXP`, `EXPORT`

Export encryption algorithms. Including 40 and 56 bits algorithms.

`EXPORT40`

40 bit export encryption algorithms.

`EXPORT56`

56 bit export encryption algorithms.

`eNULL`, `NULL`

The NULL ciphers, that is those offering no encryption. Because these offer no encryption at all and are a security risk they are disabled unless explicitly included.

`aNULL`

The cipher suites offering no authentication. This is currently the anonymous DH algorithms. These cipher suites are vulnerable to a -man in the middle- attack, and so their use is normally discouraged.

`kRSA`, `RSA`

Cipher suites using RSA key exchange.

`kEDH`

Cipher suites using ephemeral DH key agreement.

`kDHr`, `kDHd`

Cipher suites using DH key agreement and DH certificates signed by CAs with RSA and DSS keys respectively. Not implemented.

`aRSA`

Cipher suites using RSA authentication, i.e. the certificates carry RSA keys.

`aDSS`, `DSS`

Cipher suites using DSS authentication, i.e. the certificates carry DSS keys.

`aDH`

Cipher suites effectively using DH authentication, i.e. the certificates carry DH keys. Not implemented.

`kFZA`, `aFZA`, `eFZA`, `FZA`

Cipher suites using FORTEZZA key exchange, authentication, encryption or all FORTEZZA algorithms. Not implemented.

`TLSv1`, `SSLv3`, `SSLv2`

TLS v1.0, SSL v3.0 or SSL v2.0 cipher suites respectively.

`DH`

Cipher suites using DH, including anonymous DH.

`ADH`

Anonymous DH cipher suites.

`3DES`

Cipher suites using triple DES.

`DES`

Cipher suites using DES (not triple DES).

`RC4`

Cipher suites using RC4.

`RC2`

Cipher suites using RC2.

`IDEA`

Cipher suites using IDEA.

`MD5`

Cipher suites using MD5.

`SHA1`, `SHA`

Cipher suites using SHA1.

CIPHER SUITE NAMES

The following lists give the SSL or TLS cipher suites names from the relevant specification and their OpenSSL equivalents.

SSL v3.0 cipher suites

```
SSL_RSA_WITH_NULL_MD5                   NULL-MD5
SSL_RSA_WITH_NULL_SHA                   NULL-SHA
SSL_RSA_EXPORT_WITH_RC4_40_MD5          EXP-RC4-MD5
SSL_RSA_WITH_RC4_128_MD5                RC4-MD5
SSL_RSA_WITH_RC4_128_SHA                RC4-SHA
SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5      EXP-RC2-CBC-MD5
SSL_RSA_WITH_IDEA_CBC_SHA               IDEA-CBC-SHA
SSL_RSA_EXPORT_WITH_DES40_CBC_SHA       EXP-DES-CBC-SHA
SSL_RSA_WITH_DES_CBC_SHA                DES-CBC-SHA
SSL_RSA_WITH_3DES_EDE_CBC_SHA           DES-CBC3-SHA

SSL_DH_DSS_EXPORT_WITH_DES40_CBC_SHA    Not implemented.
SSL_DH_DSS_WITH_DES_CBC_SHA             Not implemented.
SSL_DH_DSS_WITH_3DES_EDE_CBC_SHA        Not implemented.
SSL_DH_RSA_EXPORT_WITH_DES40_CBC_SHA    Not implemented.
SSL_DH_RSA_WITH_DES_CBC_SHA             Not implemented.
SSL_DH_RSA_WITH_3DES_EDE_CBC_SHA        Not implemented.
SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA   EXP-EDH-DSS-DES-CBC-SHA
SSL_DHE_DSS_WITH_DES_CBC_SHA            EDH-DSS-CBC-SHA
SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA       EDH-DSS-DES-CBC3-SHA
SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA   EXP-EDH-RSA-DES-CBC-SHA
SSL_DHE_RSA_WITH_DES_CBC_SHA            EDH-RSA-DES-CBC-SHA
SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA       EDH-RSA-DES-CBC3-SHA

SSL_DH_anon_EXPORT_WITH_RC4_40_MD5      EXP-ADH-RC4-MD5
SSL_DH_anon_WITH_RC4_128_MD5            ADH-RC4-MD5
SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA   EXP-ADH-DES-CBC-SHA
SSL_DH_anon_WITH_DES_CBC_SHA            ADH-DES-CBC-SHA
SSL_DH_anon_WITH_3DES_EDE_CBC_SHA       ADH-DES-CBC3-SHA

SSL_FORTEZZA_KEA_WITH_NULL_SHA          Not implemented.
SSL_FORTEZZA_KEA_WITH_FORTEZZA_CBC_SHA  Not implemented.
SSL_FORTEZZA_KEA_WITH_RC4_128_SHA       Not implemented.
```

TLS v1.0 cipher suites

```
TLS_RSA_WITH_NULL_MD5                   NULL-MD5
TLS_RSA_WITH_NULL_SHA                   NULL-SHA
TLS_RSA_EXPORT_WITH_RC4_40_MD5          EXP-RC4-MD5
TLS_RSA_WITH_RC4_128_MD5                RC4-MD5
TLS_RSA_WITH_RC4_128_SHA                RC4-SHA
TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5      EXP-RC2-CBC-MD5
TLS_RSA_WITH_IDEA_CBC_SHA               IDEA-CBC-SHA
TLS_RSA_EXPORT_WITH_DES40_CBC_SHA       EXP-DES-CBC-SHA
TLS_RSA_WITH_DES_CBC_SHA                DES-CBC-SHA
TLS_RSA_WITH_3DES_EDE_CBC_SHA           DES-CBC3-SHA

TLS_DH_DSS_EXPORT_WITH_DES40_CBC_SHA    Not implemented.
TLS_DH_DSS_WITH_DES_CBC_SHA             Not implemented.
TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA        Not implemented.
TLS_DH_RSA_EXPORT_WITH_DES40_CBC_SHA    Not implemented.
```

```
TLS_DH_RSA_WITH_DES_CBC_SHA               Not implemented.
TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA          Not implemented.
TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA     EXP-EDH-DSS-DES-CBC-SHA
TLS_DHE_DSS_WITH_DES_CBC_SHA              EDH-DSS-CBC-SHA
TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA         EDH-DSS-DES-CBC3-SHA
TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA     EXP-EDH-RSA-DES-CBC-SHA
TLS_DHE_RSA_WITH_DES_CBC_SHA              EDH-RSA-DES-CBC-SHA
TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA         EDH-RSA-DES-CBC3-SHA

TLS_DH_anon_EXPORT_WITH_RC4_40_MD5        EXP-ADH-RC4-MD5
TLS_DH_anon_WITH_RC4_128_MD5              ADH-RC4-MD5
TLS_DH_anon_EXPORT_WITH_DES40_CBC_SHA     EXP-ADH-DES-CBC-SHA
TLS_DH_anon_WITH_DES_CBC_SHA              ADH-DES-CBC-SHA
TLS_DH_anon_WITH_3DES_EDE_CBC_SHA         ADH-DES-CBC3-SHA
```

Additional Export 1024 and other cipher suites

These ciphers can also be used in SSL v3.

```
TLS_RSA_EXPORT1024_WITH_DES_CBC_SHA       EXP1024-DES-CBC-SHA
TLS_RSA_EXPORT1024_WITH_RC4_56_SHA        EXP1024-RC4-SHA
TLS_DHE_DSS_EXPORT1024_WITH_DES_CBC_SHA   EXP1024-DHE-DSS-DES-CBC-SHA
TLS_DHE_DSS_EXPORT1024_WITH_RC4_56_SHA    EXP1024-DHE-DSS-RC4-SHA
TLS_DHE_DSS_WITH_RC4_128_SHA              DHE-DSS-RC4-SHA
```

SSL v2.0 cipher suites.

```
SSL_CK_RC4_128_WITH_MD5                   RC4-MD5
SSL_CK_RC4_128_EXPORT40_WITH_MD5          EXP-RC4-MD5
SSL_CK_RC2_128_CBC_WITH_MD5               RC2-MD5
SSL_CK_RC2_128_CBC_EXPORT40_WITH_MD5      EXP-RC2-MD5
SSL_CK_IDEA_128_CBC_WITH_MD5              IDEA-CBC-MD5
SSL_CK_DES_64_CBC_WITH_MD5                DES-CBC-MD5
SSL_CK_DES_192_EDE3_CBC_WITH_MD5          DES-CBC3-MD5
```

DESCRIPTION

The `cipherlist` command converts OpenSSL cipher lists into ordered SSL cipher preference lists. It can be used as a test tool to determine the appropriate cipherlist.

NOTES

The non-ephemeral DH modes are currently unimplemented in OpenSSL because there is no support for DH certificates.

Some compiled versions of OpenSSL may not include all the ciphers listed here because some ciphers were excluded at compile time.

EXAMPLES

Verbose listing of all OpenSSL ciphers including NULL ciphers:

```
openssl ciphers -v 'ALL:eNULL'
```

Include all ciphers except NULL and anonymous DH then sort by strength:

```
openssl ciphers -v 'ALL:!ADH:@STRENGTH'
```

Include only 3DES ciphers and then place RSA ciphers last:

```
openssl ciphers -v '3DES:+RSA'
```

SEE ALSO

Commands: *s_client*, *s_server*

Functions: *ssl*

crl

NAME

crl – CRL utility

SYNOPSIS

openssl crl [-inform PEM|DER] [-outform PEM|DER] [-text] [-in *filename*] [-out *filename*] [-noout] [-hash] [-issuer] [-lastupdate] [-nextupdate] [-CAfile *file*] [-CApath *dir*]

OPTIONS

inform DER|PEM

Specifies the input format. The DER format is DER encoded CRL structure. The PEM format (the default) is a base64 encoded version of the DER form with header and footer lines.

outform DER|PEM

Specifies the output format. The options have the same meaning as the inform option.

in *filename*

Specifies the input filename to read from or standard input if this option is not specified.

out *filename*

Specifies the output filename to write to or standard output by default.

text

Prints out the CRL in text form.

noout

Does not output the encoded version of the CRL.

hash

Outputs a hash of the issuer name. This can be used to lookup CRLs in a directory by issuer name.

issuer

Outputs the issuer name.

lastupdate

Outputs the lastupdate field.

nextupdate

Outputs the nextupdate field.

CAfile *file*

Verifies the signature on a CRL by looking up the issuing certificate in *file*

CApath *dir*

Verifies the signature on a CRL by looking up the issuing certificate in *dir*. This directory must be a standard certificate directory; that is, a hash of each subject name (using x509 hash) should be linked to each certificate.

DESCRIPTION

The `crl` command processes `CRL` files in `DER` or `PEM` format.

NOTES

The PEM CRL format uses the header and footer lines:

```
-----BEGIN X509 CRL-----
-----END X509 CRL-----
```

RESTRICTIONS

Ideally it should be possible to create a CRL using appropriate options and files.

EXAMPLES

Convert a CRL file from PEM to DER:

```
openssl crl -in crl.pem -outform DER -out crl.der
```

Output the text form of a DER encoded certificate:

```
openssl crl -in crl.der -text -noout
```

SEE ALSO

Commands: *crl2pkcs7*, *ca*, *x509*

crl2pkcs7

NAME

crl2pkcs7 – Creates a PKCS#7 structure from a CRL and certificates.

SYNOPSIS

openssl pkcs7 [-inform PEM|DER] [-outform PEM|DER] [-in *filename*] [-out *filename*] [-certfile *filename*] [-print_certs]

OPTIONS

inform DER|PEM

Specifies the CRL input format. DER format is DER encoded CRL structure.PEM (the default) is a base64 encoded version of the DER form with header and footer lines.

outform DER|PEM

Specifies the PKCS#7 structure output format. The DER format is DER encoded PKCS#7 structure. The PEM (the default) is a base64 encoded version of the DER form with header and footer lines.

in *filename*

This specifies the input filename to read a CRL from or standard input if this option is not specified.

out *filename*

Specifies the output filename to write the PKCS#7 structure to or standard output by default.

certfile *filename*

Specifies a filename containing one or more certificates in PEM format. All certificates in the file will be added to the PKCS#7 structure. This option can be used more than once to read certificates form multiple files.

nocrl

Normally a CRL is included in the output file. With this option no CRL is included in the output file and a CRL is not read from the input file.

DESCRIPTION

The crl2pkcs7 command takes an optional CRL and one or more certificates and converts them into a PKCS#7 degenerate certificates only structure.

NOTES

The output file is a PKCS#7 signed data structure containing no signers and just certificates and an optional CRL.

This utility can be used to send certificates and CAs to Netscape as part of the certificate enrollment process. This involves sending the DER encoded output as MIME type application/x-x509-user-cert.

The PEM encoded form with the header and footer lines removed can be used to install user certificates and CAs in MSIE using the Xenroll control.

EXAMPLES

Create a PKCS#7 structure from a certificate and CRL:

```
 openssl crl2pkcs7 -in crl.pem -certfile cert.pem -out p7.pem
```

Creates a PKCS#7 structure in DER format with no CRL from several different certificates:

```
 openssl crl2pkcs7 -nocrl -certfile newcert.pem
-certfile demoCA/cacert.pem -outform DER -out p7.der
```

SEE ALSO

Commands: *pkcs7*

crypto

NAME

crypto – OpenSSL cryptographic library

DESCRIPTION

The OpenSSL `crypto` library implements a wide range of cryptographic algorithms used in various Internet standards. The services provided by this library are used by the OpenSSL implementations of SSL, TLS and S/MIME, and they have also been used to implement SSH, OpenPGP, and other cryptographic standards.

The `libcrypto` consists of a number of sublibraries that implement the individual algorithms.

The functionality includes symmetric encryption, public key cryptography and key agreement, certificate handling, cryptographic hash functions and a cryptographic pseudo-random number generator.

- SYMMETRIC CIPHERS

  blowfish, *cast*, *des*, *idea*, *rc2*, *rc4*, *rc5*
- PUBLIC KEY CRYPTOGRAPHY AND KEY AGREEMENT

  dsa, *dh*, *rsa*
- CERTIFICATES

  x509, *x509v3*
- AUTHENTICATION CODES, HASH FUNCTIONS

  hmac, *md2, md4, md5, mdc2*, *ripemd160*, *sha*
- AUXILIARY FUNCTIONS

  err, *threads*, *rand*
- INPUT/OUTPUT, DATA ENCODING

  asn1, bio, *evp*, *pem*, *pkcs7*, *pkcs12*
- INTERNAL FUNCTIONS

  bn, *buffer*, *lhash*, *objects*, *stack*, *txt_db*

SEE ALSO

Files: *openssl*

Functions: *ssl*

CRYPTO_set_ex_data

NAME

CRYPTO_set_ex_data, CRYPTO_get_ex_data – Internal application specific data functions

SYNOPSIS

```
int CRYPTO_set_ex_data(
       CRYPTO_EX_DATA *r, int idx, void *arg
);
void *CRYPTO_get_ex_data(
        CRYPTO_EX_DATA *r, int idx
);
```

DESCRIPTION

Several OpenSSL structures can have application specific data attached to them. These functions are used internally by OpenSSL to manipulate application specific data attached to a specific structure.

These functions should only be used by applications to manipulate `CRYPTO_EX_DATA` structures passed to the `new_func()`, `free_func()`, and `dup_func()` callbacks:, as passed to `RSA_get_ex_new_index()`, for example.

The `CRYPTO_set_ex_data()` function is used to set application specific data. The data is supplied in the `arg` parameter and its precise meaning is up to the application.

The `CRYPTO_get_ex_data()` function is used to retrieve application specific data. The data is returned to the application. This will be the same value as supplied to a previous `CRYPTO_set_ex_data()` call.

RETURN VALUES

The `CRYPTO_set_ex_data()` function returns 1 on success or 0 on failure.

The `CRYPTO_get_ex_data()` function returns the application data or 0 on failure. Zero may also be valid application data but currently it can only fail if given an invalid `idx` parameter.

On failure an error code can be obtained from `ERR_get_error()`.

HISTORY

The `CRYPTO_set_ex_data()` and `CRYPTO_get_ex_data()` functions have been available since SSLeay 0.9.0.

SEE ALSO

Functions: *RSA_get_ex_new_index*, *DSA_get_ex_new_index*, *DH_get_ex_new_index*

d2i_DHparams

NAME

d2i_DHparams, i2d_DHparams – Purpose to be supplied.

SYNOPSIS

```
#include <openssl/dh.h>
DH *d2i_DHparams(
       DH **a, unsigned char **pp, long length
);
int i2d_DHparams(
       DH *a, unsigned char **pp
);
```

DESCRIPTION

RETURN VALUES

d2i_RSAPublicKey

NAME

d2i_RSAPublicKey, i2d_RSAPublicKey, d2i_RSAPrivateKey, i2d_RSAPrivateKey, i2d_Netscape_RSA, d2i_Netscape_RSA – Purpose to be supplied.

SYNOPSIS

```
#include <openssl/rsa.h>
RSA * d2i_RSAPublicKey(
       RSA **a, unsigned char **pp, long length
);
int i2d_RSAPublicKey(
       RSA *a, unsigned char **pp
);
RSA * d2i_RSAPrivateKey(
        RSA **a, unsigned char **pp, long length
);
int i2d_RSAPrivateKey(
       RSA *a, unsigned char **pp
);
int i2d_Netscape_RSA(
       RSA *a, unsigned char **pp, int (*cb)()
);
RSA * d2i_Netscape_RSA(
       RSA **a, unsigned char **pp, long length, int (*cb)()
);
```

DESCRIPTION

RETURN VALUES

d2i_SSL_SESSION

NAME

d2i_SSL_SESSION, i2d_SSL_SESSION – Convert SSL_SESSION object to or from ASN1 representation

SYNOPSIS

```
#include <openssl/ssl.h>
SSL_SESSION *d2i_SSL_SESSION(
        SSL_SESSION **a, unsigned char **pp, long length
);
int i2d_SSL_SESSION(
       SSL_SESSION *in, unsigned char **pp
);
```

DESCRIPTION

The `d2i_SSL_SESSION()` function transforms the external ASN1 representation of an SSL/TLS session, stored as binary data at location `pp` with length `length`, into an SSL_SESSION object.

The `i2d_SSL_SESSION()` function transforms the SSL_SESSION object `in` into the ASN1 representation and stores it into the memory location pointed to by `pp`. The length of the resulting ASN1 representation is returned. If `pp` is the NULL pointer, only the length is calculated and returned.

NOTES

The SSL_SESSION object is built from several `malloc()` parts. Therefore, it cannot be moved, copied or stored directly. In order to store session data on disk or into a database, it must be transformed into a binary ASN1 representation.

When using `d2i_SSL_SESSION()`, the SSL_SESSION object is automatically allocated.

When using `i2d_SSL_SESSION()`, the memory location pointed to by `pp` must be large enough to hold the binary representation of the session. There is no known limit on the size of the created ASN1 representation, so the necessary amount of space should be obtained by first calling `i2d_SSL_SESSION()` with `pp=NULL`, and obtain the size needed, then allocate the memory and call `i2d_SSL_SESSION()` again.

RETURN VALUES

The `d2i_SSL_SESSION()` function returns a pointer to the newly allocated SSL_SESSION object. In case of failure the NULL-pointer is returned and the error message can be retrieved from the error stack.

The `i2d_SSL_SESSION()` function returns the size of the ASN1 representation in bytes. When the session is not valid, 0 is returned and no operation is performed.

SEE ALSO

Functions: *ssl*, *SSL_CTX_sess_set_get_cb*

des

NAME

des: des_random_key, des_set_keydes_key_sche, des_set_key_checked, des_set_key_unchecked, des_set_odd_parity, des_is_weak_key, des_ecb_encrypt, des_ecb2_encrypt, des_ecb3_encrypt, des_ncbc_encrypt, des_cfb_encrypt, des_ofb_encrypt, des_pcbc_encrypt, des_cfb64_encrypt, des_ofb64_encrypt, des_xcbc_encrypt, des_ede2_cbc_encrypt, des_ede2_cfb64_encrypt, des_ede2_ofb64_encrypt, des_ede3_cbc_encrypt, des_ede3_cbcm_encrypt, des_ede3_cfb64_encrypt, des_ede3_ofb64_encrypt des_read_password, des_read_2passwords, des_read_pw_string, des_cbc_cksum, des_quad_cksum, des_string_to_key, des_string_to_2keys, des_fcryptdes_crypt, des_enc_read, des_enc_write – DES encryption

SYNOPSIS

```
#include <openssl/des.h>

void des_random_key(
       des_cblock *ret
);

int des_set_key(
       const_des_cblock *key, des_key_schedule schedule
);

int des_key_sched(
       const_des_cblock *key, des_key_schedule schedule
);

int des_set_key_checked(
        const_des_cblock *key, des_key_schedule schedule
);

void des_set_key_unchecked(
        const_des_cblock *key, des_key_schedule schedule
);

void des_set_odd_parity(
        des_cblock *key
);

int des_is_weak_key(
       const_des_cblock *key
);

void des_ecb_encrypt(
       const_des_cblock *input, des_cblock *output, des_key_schedule ks, int enc
);

void des_ecb2_encrypt(
       const_des_cblock *input, des_cblock *output, des_key_schedule ks1,
       des_key_schedule ks2, int enc
);
```

```
void des_ecb3_encrypt(
        const_des_cblock *input, des_cblock *output, des_key_schedule ks1,
        des_key_schedule ks2, des_key_schedule ks3, int enc
);
void des_ncbc_encrypt(
        const unsigned char *input, unsigned char *output, long length, des_key_schedule
        schedule, des_cblock *ivec, int enc
);
void des_cfb_encrypt(
        const unsigned char *in, unsigned char *out, int numbits, long length,
        des_key_schedule schedule, des_cblock *ivec, int enc
);
void des_ofb_encrypt(
        const unsigned char *in, unsigned char *out, int numbits, long length,
        des_key_schedule schedule, des_cblock *ivec
);
void des_pcbc_encrypt(
        const unsigned char *input, unsigned char *output, long length, des_key_schedule
        schedule, des_cblock *ivec, int enc
);
void des_cfb64_encrypt(
        const unsigned char *in, unsigned char *out, long length, des_key_schedule
        schedule, des_cblock *ivec, int *num, int enc
);
void des_ofb64_encrypt(
        const unsigned char *in, unsigned char *out, long length, des_key_schedule
        schedule, des_cblock *ivec, int *num
);
void des_xcbc_encrypt(
        const unsigned char *input, unsigned char *output, long length, des_key_schedule
        schedule, des_cblock *ivec, const_des_cblock *inw, const_des_cblock *outw, int
        enc
);
void des_ede2_cbc_encrypt(
         const unsigned char *input, unsigned char *output, long length, des_key_schedule
        ks1, des_key_schedule ks2, des_cblock *ivec, int enc
);
void des_ede2_cfb64_encrypt(
         const unsigned char *in, unsigned char *out, long length, des_key_schedule ks1,
        des_key_schedule ks2, des_cblock *ivec, int *num, int enc
);
void des_ede2_ofb64_encrypt(
         const unsigned char *in, unsigned char *out, long length, des_key_schedule ks1,
        des_key_schedule ks2, des_cblock *ivec, int *num
);
```

```
void des_ede3_cbc_encrypt(
        const unsigned char *input, unsigned char *output, long length, des_key_schedule
        ks1, des_key_schedule ks2, des_key_schedule ks3, des_cblock *ivec, int enc)
);
void des_ede3_cbcm_encrypt(
        const unsigned char *in, unsigned char *out, long length, des_key_schedule ks1,
        des_key_schedule ks2, des_key_schedule ks3, des_cblock *ivec1, des_cblock *ivec2,
        int enc
);
void des_ede3_cfb64_encrypt(
        const unsigned char *in, unsigned char *out, long length, des_key_schedule ks1,
        des_key_schedule ks2, des_key_schedule ks3, des_cblock *ivec, int *num, int enc
);
void des_ede3_ofb64_encrypt(
        const unsigned char *in, unsigned char *out, long length, des_key_schedule ks1,
        des_key_schedule ks2, des_key_schedule ks3, des_cblock *ivec, int *num
);
int des_read_password(
        des_cblock *key, const char *prompt, int verify
);
int des_read_2passwords(
        des_cblock *key1, des_cblock *key2, const char *prompt, int verify
);
int des_read_pw_string(
        char *buf, int length, const char *prompt, int verify
);
DES_LONG des_cbc_cksum(
        const unsigned char *input, des_cblock *output, long length, des_key_schedule
        schedule, const_des_cblock *ivec
);
DES_LONG des_quad_cksum(
        const unsigned char *input, des_cblock output[], long length, int out_count,
        des_cblock *seed
);
void des_string_to_key(
        const char *str, des_cblock *key
);
void des_string_to_2keys(
        const char *str, des_cblock *key1, des_cblock *key2
);
char *des_fcrypt(
        const char *buf, const char *salt, char *ret
);
```

```
char *des_crypt(
        const char *buf, const char *salt
);
char *crypt(
        const char *buf, const char *salt
);
int des_enc_read(
        int fd, void *buf, int len, des_key_schedule sched, des_cblock *iv
);
int des_enc_write(
        int fd, const void *buf, int len, des_key_schedule sched, des_cblock *iv
);
```

DESCRIPTION

This library contains a fast implementation of the DES encryption algorithm.

There are two phases to the use of DES encryption. The first is the generation of a `des_key_schedule` from a key; the second is the actual encryption. A DES key is of type `des_cblock`. This type consists of 8 bytes with odd parity. The least significant bit in each byte is the parity bit. The key schedule is an expanded form of the key; it is used to speed the encryption process.

The `des_random_key()` generates a random key. The PRNG must be seeded prior to using this function (see *rand*; for backward compatibility the `des_random_seed()` function is available as well). If the PRNG could not generate a secure key, 0 is returned. In earlier versions of the library, `des_random_key()` did not generate secure keys.

Before a DES key can be used, it must be converted into the architecture dependent `des_key_schedule` via the `des_set_key_checked()` or `des_set_key_unchecked()` functions.

The `des_set_key_checked()` function will check that the key passed is of odd parity and is not a weak or semi-weak key. If the parity is wrong, then -1 is returned. If the key is a weak key, then -2 is returned. If an error is returned, the key schedule is not generated.

The `des_set_key()` function (called `des_key_sched()` in the MIT library) works like `des_set_key_checked()` if the `des_check_key` flag is non-zero; otherwise, it works like `des_set_key_unchecked()`. These functions are available for compatibility; we recommend you use a function that does not depend on a global variable.

The `des_set_odd_parity()` function (called `des_fixup_key_parity()` in the MIT library) sets the parity of the passed `key` to odd.

The `des_is_weak_key()` function returns 1 is the passed key is a weak key, 0 if it is ok. The probability that a randomly generated key is weak is 1/2^52.

The following routines mostly operate on an input and output stream of `des_cblock`:

- The `des_ecb_encrypt()` function is the basic DES encryption routine that encrypts or decrypts a single 8-byte `des_cblock` in electronic code book (ECB) mode. It always transforms the input data, pointed to by input, into the output data, pointed to by the output argument. If the encrypt argument is non-zero (DES_ENCRYPT), the input (cleartext) is encrypted in to the output (ciphertext) using the key_schedule specified by the schedule argument, previously set via `des_set_key`. If encrypt is zero (DES_DECRYPT), the input (now ciphertext) is decrypted into the output (now cleartext). Input and output may overlap. The `des_ecb_encrypt()` function does not return a value.

- The des_ecb3_encrypt() function encrypts and decrypts the input block by using three-key Triple-DES encryption in ECB mode. This involves encrypting the input with ks1, decrypting with the key schedule ks2, and then encrypting with ks3. This routine greatly reduces the chances of brute force breaking of DES and has the advantage if ks1, ks2 and ks3 are the same. It is equivalent to encryption using ECB mode and ks1 as the key.
- The des_ecb2_encrypt() macro is provided to perform two-key Triple-DES encryption by using ks1 for the final encryption.
- The des_ncbc_encrypt() function encrypts and decrypts using the cipher-block-chaining (CBC) mode of DES. If the encrypt argument is non-zero, the routine cipher-block-chain encrypts the cleartext data pointed to by the input argument into the ciphertext pointed to by the output argument, using the key schedule provided by the schedule argument, and initialization vector provided by the ivec argument. If the length argument is not an integral multiple of eight bytes, the last block is copied to a temporary area and zero filled. The output is always an integral multiple of eight bytes.
- The des_xcbc_encrypt() function is RSA's DESX mode of DES. It uses inw and outw to whiten the encryption. The inw and outw are secret (unlike the iv) and are part of the key. So the key is sort of 24 bytes. This is much better than CBC DES.
- The des_ede3_cbc_encrypt() function implements outer triple CBC DES encryption with three keys. This means that each DES operation inside the CBC mode is really an C=E(ks3,D(ks2,E(ks1,M))). This mode is used by SSL.
- The des_ede2_cbc_encrypt() macro implements two-key Triple-DES by reusing ks1 for the final encryption. C=E(ks1,D(ks2,E(ks1,M))). This form of Triple-DES is used by the RSAREF library.
- The des_pcbc_encrypt() function encrypts and decrypts using the propagating cipher block chaining mode used by Kerberos v4. Its parameters are the same as des_ncbc_encrypt().
- The des_cfb_encrypt() function encrypts and decrypts using cipher feedback mode. This method takes an array of characters as input and outputs and array of characters. It does not require any padding to 8 character groups. The ivec variable is changed and the new changed value needs to be passed to the next call to this function. Since this function runs a complete DES ECB encryption per numbits, this function is only suggested for use when sending small numbers of characters.
- The des_cfb64_encrypt() function implements CFB mode of DES with 64-bit feedback. This is useful because this routine will allow you to encrypt an arbitrary number of bytes, no 8-byte padding. Each call to this routine will encrypt the input bytes to output and then update ivec and num. The num shows where you are through ivec.
- The des_ede3_cfb64_encrypt() and des_ede2_cfb64_encrypt() functions are the same as the des_cfb64_encrypt() function except that Triple-DES is used.
- The des_ofb_encrypt() function encrypts using output feedback mode. This method takes an array of characters as input and outputs and array of characters. It does not require any padding to 8-character groups. The ivec variable is changed and the new changed value needs to be passed to the next call to this function. Since this function runs a complete DES ECB encryption per numbits, we recommend using this function only when sending small numbers of characters.
- The des_ofb64_encrypt() function is the same as the des_cfb64_encrypt() function using Output Feed Back mode.
- The des_ede3_ofb64_encrypt() and des_ede2_ofb64_encrypt() functions are the same as des_ofb64_encrypt() using Triple-DES.

The following functions are included in the DES library for compatibility with the MIT Kerberos library. The des_read_pw_string() function is also available under the name EVP_read_pw_string().

- The `des_read_pw_string()` function writes the string specified by prompt to standard output, turns echo off and reads in input string from the terminal. The string is returned in `buf`, which must have space for at least `length` bytes. If `verify` is set, the user is asked for the password twice. Unless the two copies match, an error is returned. A return code of -1 indicates a system error, 1 failure due to use interaction, and 0 is success.
- The `des_read_password()` function does the same and converts the password to a DES key by calling `des_string_to_key()`; the `des_read_2password()` function operates in the same way as `des_read_password()` except that it generates two keys by using the `des_string_to_2key()` function. The `des_string_to_key()` function is available for backward compatibility with the MIT library. New applications should use a cryptographic hash function. The same applies for the `des_string_to_2key()` function.
- The `des_cbc_cksum()` function produces an 8-byte checksum based on the input stream (via CBC encryption). The last 4 bytes of the checksum are returned and the complete 8 bytes are placed in `output`. This function is used by Kerberos v4. Other applications should use `EVP_DigestInit()` etc. instead.
- The `des_quad_cksum()` function is a Kerberos v4 function. It returns a 4-byte checksum from the input bytes. The algorithm can be iterated over the input, depending on `out_count`, 1, 2, 3 or 4 times. If output is non-NULL, the 8 bytes generated by each pass are written into output.

The following are DES-based transformations:

- The `des_fcrypt()` function is a fast version of the Unix `crypt()` function. This version takes only a small amount of space relative to other fast `crypt()` implementations. This is different from the normal crypt in that the third parameter is the buffer that the return value is written into. It needs to be at least 14 bytes long. This function is thread safe, unlike the normal crypt.
- The `des_crypt()` function is a faster replacement for the normal system `crypt()`. This function calls `des_fcrypt()` with a static array passed as the third parameter. This emulates the normal non-thread safe semantics of `crypt()`.
- The `des_enc_write()` function writes `len` bytes to file descriptor `fd` from buffer `buf`. The data is encrypted via `pcbc_encrypt` (default) using `sched` for the key and `iv` as a starting vector. The actual data send down `fd` consists of 4 bytes (in network byte order) containing the length of the following encrypted data. The encrypted data then follows, padded with random data out to a multiple of 8 bytes.
- The `des_enc_read()` function is used to read `len` bytes from file descriptor `fd` into buffer `buf`. The data being read from `fd` is assumed to have come from `des_enc_write()` and is decrypted using `sched` for the key schedule and `iv` for the initial vector.

NOTE The data format used by `des_enc_write()` and `des_enc_read()` has a cryptographic weakness: When asked to write more than MAXWRITE bytes, `des_enc_write()` will split the data into several chunks that are all encrypted using the same IV. We do not recommend using these functions unless you are sure you know what you do. They cannot handle non-blocking sockets. The `des_enc_read()` function uses an internal state and cannot be used on multiple files.

- The `des_rw_mode` specifies the encryption mode to use with the `des_enc_read()` and `des_end_write()` functions. If it is set to `DES_PCBC_MODE` (the default), `des_pcbc_encrypt` is used. If it is set to `DES_CBC_MODE`, `des_cbc_encrypt` is used.

NOTES

Single-key DES is insecure due to its short key size. ECB mode is not suitable for most applications; see *des_modes*.

The *evp* library provides higher-level encryption functions.

RESTRICTIONS

The `des_3cbc_encrypt()` function is flawed and must not be used in applications.

The `des_cbc_encrypt()` function does not modify `ivec`; use the `des_ncbc_encrypt()` function instead.

The `des_cfb_encrypt()` and `des_ofb_encrypt()` functions operate on input of 8 bits. What this means is that if you set numbits to 12, and length to 2, the first 12 bits will come from the first input byte and the low half of the second input byte. The second 12 bits will have the low 8 bits taken from the 3rd input byte and the top 4 bits taken from the fourth input byte. The same holds for output. This function has been implemented this way because most people will be using a multiple of 8.

The `des_read_pw_string()` function is the most machine/OS dependent function and normally generates the most problems when porting this code.

The `des` library was written to be source code compatible with the MIT Kerberos library. It conforms to ANSI X3.106.

HISTORY

The `des_cbc_cksum()`, `des_cbc_encrypt()`, `des_ecb_encrypt()`, `des_is_weak_key()`, `des_key_sched()`, `des_pcbc_encrypt()`, `des_quad_cksum()`, `des_random_key()`, `des_read_password()`, and `des_string_to_key()` functions are available in the MIT Kerberos library; the `des_check_key_parity()`, `des_fixup_key_parity()`, and `des_is_weak_key()` functions are available in newer versions of that library.

The `des_set_key_checked()` and `des_set_key_unchecked()` functions were added in OpenSSL 0.9.5.

The `des_generate_random_block()`, `des_init_random_number_generator()`, `des_new_random_key()`, `des_set_random_generator_seed()`, `des_set_sequence_number()`, and `des_rand_data()` functions are used in newer versions of Kerberos but are not implemented here.

The `des_random_key()` function generated cryptographically weak random data in SSLeay and in OpenSSL prior version 0.9.5, as well as in the original MIT library.

Author is Eric Young (eay@cryptsoft.com). Modified for the OpenSSL project (http://www.openssl.org).

SEE ALSO

Functions: *crypt*, *evp*, *rand*

Files: *des_modes*

des_modes

NAME

des_modes – Variants of DES and other crypto algorithms of OpenSSL

DESCRIPTION

Several crypto algorithms for OpenSSL can be used in a number of modes. Those are used for using block ciphers in a way similar to stream ciphers, among other things.

Electronic Codebook Mode (ECB)

Normally, this is found as the `algorithm_ecb_encrypt()` function.

- 64 bits are enciphered at a time.
- The order of the blocks can be rearranged without detection.
- The same plaintext block always produces the same ciphertext block (for the same key) making it vulnerable to a dictionary attack.
- An error will only affect one ciphertext block.

Cipher Block Chaining Mode (CBC)

Normally, this is found as the `algorithm_cbc_encrypt()`function. Be aware that `des_cbc_encrypt()` is not really DES CBC (it does not update the IV); use the `des_ncbc_encrypt()` function instead.

- A multiple of 64 bits are enciphered at a time.
- The CBC mode produces the same ciphertext whenever the same plaintext is encrypted using the same key and starting variable.
- The chaining operation makes the ciphertext blocks dependent on the current and all preceding plaintext blocks and therefore blocks can not be rearranged.
- The use of different starting variables prevents the same plaintext enciphering to the same ciphertext.
- An error will affect the current and the following ciphertext blocks.

Cipher Feedback Mode (CFB)

Normally, this is found as the `algorithm_cfb_encrypt()` function.

- A number of bits (j) <= 64 are enciphered at a time.
- The CFB mode produces the same ciphertext whenever the same plaintext is encrypted using the same key and starting variable.
- The chaining operation makes the ciphertext variables dependent on the current and all preceding variables and therefore j-bit variables are chained together and can not be rearranged.
- The use of different starting variables prevents the same plaintext enciphering to the same ciphertext.
- The strength of the CFB mode depends on the size of k (maximal if j == k).
- Selection of a small value for j will require more cycles through the encipherment algorithm per unit of plaintext and thus cause greater processing overheads.
- Only multiples of j bits can be enciphered.

- An error will affect the current and the following ciphertext variables.

Output Feedback Mode (OFB)

Normally, this is found as the *algorithm*_ofb_encrypt() function.

- A number of bits (j) <= 64 are enciphered at a time.
- The OFB mode produces the same ciphertext whenever the same plaintext enciphered using the same key and starting variable. More over, in the OFB mode the same key stream is produced when the same key and start variable are used. Consequently, for security reasons a specific start variable should be used only once for a given key.
- The absence of chaining makes the OFB more vulnerable to specific attacks.
- The use of different start variables values prevents the same plaintext enciphering to the same ciphertext, by producing different key streams.
- Selection of a small value for j will require more cycles through the encipherment algorithm per unit of plaintext and thus cause greater processing overheads.
- Only multiples of j bits can be enciphered.
- OFB mode of operation does not extend ciphertext errors in the resultant plaintext output. Every bit error in the ciphertext causes only one bit to be in error in the deciphered plaintext.
- OFB mode is not self-synchronizing. If the two operation of encipherment and decipherment get out of synchronism, the system needs to be reinitialized.
- Each reinitialization should use a value of the start variable different from the start variable values used before with the same key. The reason for this is that an identical bit stream would be produced each time from the same parameters. This would be susceptible to a known plaintext attack.

Triple ECB Mode

Normally, this is found as the *algorithm*_ecb3_encrypt() function .

- Encrypt with key1, decrypt with key2 and encrypt with key3 again.
- As for ECB encryption but increases the key length to 168 bits. There are theoretic attacks that can be used that make the effective key length 112 bits, but this attack also requires 2^56 blocks of memory, not very likely, even for the NSA.
- If both keys are the same it is equivalent to encrypting once with just one key.
- If the first and last key are the same, the key length is 112 bits. There are attacks that could reduce the key space to 55 bits, but it requires 2^56 blocks of memory.
- If all 3 keys are the same, this is the same as normal ecb mode.

Triple CBC Mode

Normally, this is found as the *algorithm*_ede3_cbc_encrypt() function .

- Encrypt with key1, decrypt with key2 and then encrypt with key3.
- As for CBC encryption but increases the key length to 168 bits with the same restrictions as for triple ecb mode.

HISTORY

Much of this text was written by Eric Young in his original documentation for SSLeay, the predecessor of OpenSSL. In turn, he attributed it to:

```
AS 2805.5.2
Australian Standard
Electronic funds transfer - Requirements for interfaces,
Part 5.2: Modes of operation for an n-bit block cipher algorithm
Appendix A
```

SEE ALSO

Functions: *blowfish*, *des*, *idea*, *rc2*

dgst

NAME

dgst, md5, md4, md2, sha1, sha, mdc2, ripemd160 – Message digests

SYNOPSIS

openssl dgst [-md5|-md4|-md2|-sha1|-sha|-mdc2|-ripemd160|-dss1] [-c] [-d] [-hex] [-binary] [-out *filename*] [-sign *filename*] [-verify *filename*] [-prverify *filename*] [-signature *filename*] [-rand *file(s)*] [-*file...*]

OPTIONS

c

Prints out the digest in two digit groups separated by colons, only relevant if hex format output is used.

d

Prints out BIO debugging information.

hex

Digest is to be output as a hex dump. This is the default case for a typical digest as opposed to a digital signature.

binary

Outputs the digest or signature in binary form.

out *filename*

Filename to output to, or standard output by default.

sign *filename*

Digitally signs the digest using the private key in *filename*.

verify *filename*

Verifies the signature using the public key in *filename*. The output is either Verification OK or Verification Failure.

prverify *filename*

Verifies the signature using the private key in *filename*.

signature *filename*

The actual signature to verify.

rand *file(s)*

A file or files containing random data used to seed the random number generator, or an EGD socket. (See *RAND_egd*.) Multiple files can be specified separated by an OS-dependent character. The separator is a semicolon (;) for MS-Windows®, a comma (,) for OpenVMS, and a colon (:) for all others.

file...

File or files to digest. If no files are specified then standard input is used.

DESCRIPTION

The digest functions output the message digest of a supplied file or files in hexadecimal form. They can also be used for digital signing and verification.

NOTES

The digest of choice for all new applications is `SHA1`. However, other digests are still widely used.

If you wish to sign or verify data using the DSA algorithm then the `dss1` digest must be used.

A source of random numbers is required for certain signing algorithms, in particular DSA.

The signing and verify options should only be used if a single file is being signed or verified.

dh

NAME

dh – Diffie-Hellman key agreement

SYNOPSIS

```
#include <openssl/dh.h>
#include <openssl/engine.h>

DH *DH_new(
        void
);

void DH_free(
        DH *dh
);

intDH_size(
        DH *dh
);

DH * DH_generate_parameters(
         int prime_len, int generator, void (*callback)(int, int, void *), void *cb_arg);
        intDH_check(DH *dh, int *codes
);

intDH_generate_key(
        DH *dh
);

intDH_compute_key(
        unsigned char *key, BIGNUM *pub_key, DH *dh
);

void DH_set_default_openssl_method(
        DH_METHOD *meth
);

DH_METHOD *DH_get_default_openssl_method(
        void
);

intDH_set_method(
        DH *dh, ENGINE *engine
);

DH *DH_new_method(
        ENGINE *engine
);

DH_METHOD *DH_OpenSSL(
        void
```

```
);
intDH_get_ex_new_index(
        long argl, char *argp, int (*new_func)(), int (*dup_func)(), void (*free_func)()
);
intDH_set_ex_data(
        DH *d, int idx, char *arg
);
char *DH_get_ex_data(
        DH *d, int idx
);
 DH * d2i_DHparams(
        DH **a, unsigned char **pp, long length
);
inti2d_DHparams(
        DH *a, unsigned char **pp
);
intDHparams_print_fp(
        FILE *fp, DH *x
);
intDHparams_print(
        BIO *bp, DH *x
);
```

DESCRIPTION

These functions implement the Diffie-Hellman key agreement protocol. The generation of shared DH parameters is described in *DH_generate_parameters*. See *DH_generate_key* for a description of how to perform a key agreement.

The `DH` structure consists of several BIGNUM components:

```
struct
       {
       BIGNUM *p;// prime number (shared)
       BIGNUM *g;// generator of Z_p (shared)
       BIGNUM *priv_key;// private DH value x
       BIGNUM *pub_key;// public DH value g^x
       // ...
       };
DH
```

SEE ALSO

Commands: *dHParam*

Functions: *bn*, *dsa*, *err*, *rand*, *rsa*, *DH_set_method*, *DH_new*, *DH_get_ex_new_index*, *DH_generate_parameters*, *DH_compute_key*, *d2i_DHparams*, *RSA_print*

DH_generate_key

NAME

DH_generate_key, DH_compute_key – Perform Diffie-Hellman key exchange

SYNOPSIS

```
#include <openssl/dh.h>
int DH_generate_key(
        DH *dh
);
int DH_compute_key(
        unsigned char *key, BIGNUM *pub_key, DH *dh
);
```

DESCRIPTION

The `DH_generate_key()` function performs the first step of a Diffie-Hellman key exchange by generating private and public DH values. By calling `DH_compute_key()`, these are combined with the other party's public value to compute the shared key.

The `DH_generate_key()` function expects `dh` to contain the shared parameters `dh->p` and `dh->g`. It generates a random private DH value unless `dh->priv_key` is already set, and computes the corresponding public value `dh->pub_key`, which can then be published.

The `DH_compute_key()` function computes the shared secret from the private DH value in `dh` and the other party's public value in `pub_key` and stores it in `key`. The `key` must point to `DH_size(dh)` bytes of memory.

RETURN VALUES

The `DH_generate_key()` function returns 1 on success, 0 otherwise.

The `DH_compute_key()` function returns the size of the shared secret on success, -1 on error.

The error codes can be obtained from `ERR_get_error()`.

HISTORY

The `DH_generate_key()` and `DH_compute_key()` functions are available in all versions of SSLeay and OpenSSL.

SEE ALSO

Functions: *dh*, *err*, *rand*, *DH_size*

DH_generate_parameters

NAME

DH_generate_parameters, DH_check – Generate and check Diffie-Hellman parameters

SYNOPSIS

```
#include <openssl/dh.h>

DH *DH_generate_parameters(

        int prime_len, int generator, void (*callback)(int, int, void *), void *cb_arg

);

int DH_check(

        DH *dh, int *codes

);
```

DESCRIPTION

The `DH_generate_parameters()` function generates Diffie-Hellman parameters that can be shared among a group of users, and returns them in a newly allocated `DH` structure. The pseudo-random number generator must be seeded prior to calling `DH_generate_parameters()`.

The `prime_len` is the length in bits of the safe prime to be generated. The `generator` is a small number > 1, typically 2 or 5.

A callback function may be used to provide feedback about the progress of the key generation. If `callback` is not `NULL`, it will be called as described in `BN_generate_prime()` while a random prime number is generated, and when a prime has been found, `callback(3, 0, cb_arg)` is called.

`DH_check()` validates Diffie-Hellman parameters. It checks that `p` is a safe prime, and that `g` is a suitable generator. In the case of an error, the bit flags `DH_CHECK_P_NOT_SAFE_PRIME` or `DH_NOT_SUITABLE_GENERATOR` are set in `*codes`. `DH_UNABLE_TO_CHECK_GENERATOR` is set if the generator cannot be checked, meaning it does not equal 2 or 5.

NOTES

The `DH_generate_parameters()` function may run for several hours before finding a suitable prime.

The parameters generated by `DH_generate_parameters()` are not to be used in signature schemes.

RESTRICTIONS

If `generator` is not 2 or 5, `dh->g=generator` is not a usable generator.

RETURN VALUES

The `DH_generate_parameters()` function returns a pointer to the DH structure, or NULL if the parameter generation fails. The error codes can be obtained from `ERR_get_error()`.

The `DH_check()` function returns 1 if the check could be performed, 0 otherwise.

HISTORY

The `DH_check()` function is available in all versions of SSLeay and OpenSSL. The `cb_arg` argument to `DH_generate_parameters()` was added in SSLeay 0.9.0.

In versions before OpenSSL 0.9.5, `DH_CHECK_P_NOT_STRONG_PRIME` is used instead of `DH_CHECK_P_NOT_SAFE_PRIME`.

SEE ALSO

Functions: *dh*, *err*, *rand*, *DH_free*

DH_get_ex_new_index

NAME

DH_get_ex_new_index, DH_set_ex_data, DH_get_ex_data – Add application specific data to DH structures

SYNOPSIS

```
#include <openssl/dh.h>

int DH_get_ex_new_index(
         long argl, void *argp, CRYPTO_EX_new *new_func, CRYPTO_EX_dup *dup_func,
        CRYPTO_EX_free *free_func
);

int DH_set_ex_data(
        DH *d, int idx, void *arg
);

char *DH_get_ex_data(
        DH *d, int idx
);
```

DESCRIPTION

These functions handle application specific data in DH structures. Their usage is identical to that of `RSA_get_ex_new_index()`, `RSA_set_ex_data()`, and `RSA_get_ex_data()` as described in `RSA_get_ex_new_index()`.

HISTORY

The `DH_get_ex_new_index()`, `DH_set_ex_data()`, and `DH_get_ex_data()` functions are available since OpenSSL 0.9.5.

SEE ALSO

Functions: *RSA_get_ex_new_index*, *dh*

DH_new

NAME

DH_new, DH_free – Allocate and free DH objects

SYNOPSIS

```
#include <openssl/dh.h>
DH* DH_new(
        void
);
void DH_free(
        DH *dh
);
```

DESCRIPTION

The `DH_new()` function allocates and initializes a `DH` structure.

The `DH_free()` function frees the `DH` structure and its components. The values are erased before the memory is returned to the system.

RETURN VALUES

If the allocation fails, the `DH_new()` function returns `NULL` and sets an error code that can be obtained from `ERR_get_error()`. Otherwise it returns a pointer to the newly allocated structure.

The `DH_free()` function returns no value.

HISTORY

The `DH_new()` and `DH_free()` functions are available in all versions of SSLeay and OpenSSL.

SEE ALSO

Functions: *dh*, *err*, *DH_generate_parameters*, *DH_generate_key*

DH_set_method

NAME

DH_set_method, DH_set_default_openssl_method, DH_get_default_openssl_method, DH_new_method, DH_OpenSSL – Select DH method

SYNOPSIS

```
#include <openssl/dh.h>
#include <openssl/engine.h>

void DH_set_default_openssl_method(
        DH_METHOD *meth
);

DH_METHOD *DH_get_default_openssl_method(
        void
);

int DH_set_method(
        DH *dh, ENGINE *engine
);

DH *DH_new_method(
        ENGINE *engine
);

DH_METHOD *DH_OpenSSL(
        void
);
```

DESCRIPTION

A `DH_METHOD` specifies the functions that OpenSSL uses for Diffie-Hellman operations. By modifying the method, alternative implementations such as hardware accelerators may be used.

Initially, the default is to use the OpenSSL internal implementation. The `DH_OpenSSL()` function returns a pointer to that method.

The `DH_set_default_openssl_method()` function makes `meth` the default method for all DH structures created later. This is true only while the default engine for Diffie-Hellman operations remains as openssl. Engines provide an encapsulation for implementations of one or more algorithms, and all the DH functions mentioned here operate within the scope of the default openssl engine.

The `DH_get_default_openssl_method()` function returns a pointer to the current default method for the openssl engine.

The `DH_set_method()` function selects `engine` as the engine that will be responsible for all operations using the structure `dh`. If this function completes successfully, then the `dh` structure will have its own functional reference of `engine`, so the caller should remember to free their own reference to `engine` when finished with it. An engine's DH_METHOD can be retrieved (or set) by the `ENGINE_get_DH()` or `ENGINE_set_DH()` functions.

The `DH_new_method()` function allocates and initializes a DH structure so that `engine` will be used for the DH operations. If `engine` is NULL, the default engine for Diffie-Hellman operations is used.

DH_METHOD Structure

```
 typedef struct dh_meth_st
 {

     /* name of the implementation */

const char *name;

     /* generate private and public DH values for key agreement */
        int (*generate_key)(DH *dh);

     /* compute shared secret */
        int (*compute_key)(unsigned char *key, BIGNUM *pub_key, DH *dh);

     /* compute r = a ^ p mod m (May be NULL for some implementations) */
        int (*bn_mod_exp)(DH *dh, BIGNUM *r, BIGNUM *a, const BIGNUM *p,
                                const BIGNUM *m, BN_CTX *ctx,
                                BN_MONT_CTX *m_ctx);

     /* called at DH_new */
        int (*init)(DH *dh);

     /* called at DH_free */
        int (*finish)(DH *dh);
        int flags;
        char *app_data; /* ?? */

 } DH_METHOD;
```

RETURN VALUES

The `DH_OpenSSL()` and `DH_get_default_method()` functions return pointers to the respective DH_METHODs.

The `DH_set_default_openssl_method()` function returns no value.

The `DH_set_method()` function returns non-zero if the engin associated with `dh` was successfully changed to `engine`.

The `DH_new_method()` function returns NULL and sets an error code that can be obtained by `ERR_get_error()` if the allocation fails. Otherwise it returns a pointer to the newly allocated structure.

HISTORY

The `DH_set_default_method()`, `DH_get_default_method()`, `DH_set_method()`, `DH_new_method()`, and `DH_OpenSSL()` functions were added in OpenSSL 0.9.4.

The `DH_set_default_openssl_method()` and `DH_get_default_openssl_method()` replaced `DH_set_default_method()` and `DH_get_default_method()` respectively, and `DH_set_method()` and `DH_new_method()` were altered to use `ENGINE`s rather than `DH_METHOD`s during development of OpenSSL 0.9.6.

DH_size

NAME

DH_size – Get Diffie-Hellman prime size

SYNOPSIS

```
#include <openssl/dh.h>
int DH_size(
       DH *dh
);
```

DESCRIPTION

This `DH_size()` function returns the Diffie-Hellman size in bytes. It can be used to determine how much memory must be allocated for the shared secret computed by `DH_compute_key()`.

The `dh->p` must not be `NULL`.

RETURN VALUE

The size in bytes.

HISTORY

The `DH_size()` function is available in all versions of SSLeay and OpenSSL.

SEE ALSO

Functions: *dh*, *DH_generate_key*

dsa

NAME

dsa – Digital Signature Algorithm

SYNOPSIS

```
#include <openssl/dsa.h>
#include <openssl/engine.h>

DSA * DSA_new(
        void
);

void DSA_free(
        DSA *dsa
);

int DSA_size(
        DSA *dsa
);

DSA * DSA_generate_parameters(
         int bits, unsigned char *seed, int seed_len, int *counter_ret, unsigned long
        *h_ret, void (*callback)(int, int, void*), void *cb_arg
);

DH * DSA_dup_DH(
        DSA *r
);

intDSA_generate_key(
        DSA *dsa
);

intDSA_sign(
        int dummy, const unsigned char *dgst, int len, unsigned char *sigret, unsigned
        int *siglen, DSA *dsa); intDSA_sign_setup(DSA *dsa, BN_CTX *ctx, BIGNUM **kinvp,
        BIGNUM **rp
);

intDSA_verify(
        int dummy, const unsigned char *dgst, int len, unsigned char *sigbuf, int siglen,
        DSA *dsa
);

void DSA_set_default_openssl_method(
        DSA_METHOD *meth
);

DSA_METHOD *DSA_get_default_openssl_method(
        void
);
```

```
int DSA_set_method(
        DSA *dsa, ENGINE *engine
);
DSA *DSA_new_method(
        ENGINE *engine
);
DSA_METHOD *DSA_OpenSSL(
         void
);
int DSA_get_ex_new_index(
         long argl, char *argp, int (*new_func)(), int (*dup_func)(), void (*free_func)()
);
int DSA_set_ex_data(
        DSA *d, int idx, char *arg
);
char *DSA_get_ex_data(
        DSA *d, int idx
);
DSA_SIG *DSA_SIG_new(
        void
);
voidDSA_SIG_free(
        DSA_SIG *a
);
inti2d_DSA_SIG(
        DSA_SIG *a, unsigned char **pp
);
DSA_SIG *d2i_DSA_SIG(
        DSA_SIG **v, unsigned char **pp, long length
);
DSA_SIG *DSA_do_sign(
        const unsigned char *dgst, int dlen, DSA *dsa
);
intDSA_do_verify(
        onst unsigned char *dgst, int dgst_len, DSA_SIG *sig, DSA *dsa
);
DSA * d2i_DSAPublicKey(
        DSA **a, unsigned char **pp, long length
);
 DSA *d2i_DSAPrivateKey(
```

```
        DSA **a, unsigned char **pp, long length
);
DSA * d2i_DSAparams(
        DSA **a, unsigned char **pp, long length
);
inti2d_DSAPublicKey(
        DSA *a, unsigned char **pp
);
inti2d_DSAPrivateKey(
        DSA *a, unsigned char **pp
);
inti2d_DSAparams(
        DSA *a,unsigned char **pp
);
intDSAparams_print(
        BIO *bp, DSA *x
);
intDSAparams_print_fp(
        FILE *fp, DSA *x
);
intDSA_print(
        BIO *bp, DSA *x, int off
);
intDSA_print_fp(
        FILE *bp, DSA *x, int off
);
```

DESCRIPTION

These functions implement the Digital Signature Algorithm (DSA). The generation of shared DSA parameters is described in *DSA_generate_parameters*; *DSA_generate_key* describes how to generate a signature key. Signature generation and verification are described in *DSA_sign*.

The `DSA` structure consists of several BIGNUM components.

```
struct
       {
       BIGNUM *p;// prime number (public)
       BIGNUM *q;// 160-bit subprime, q | p-1 (public)
       BIGNUM *g;// generator of subgroup (public)
       BIGNUM *priv_key;// private key x
       BIGNUM *pub_key;// public key y = g^x
       // ...
       }
DSA;
```

In public keys, `priv_key` is NULL.

DSA conforms to US Federal Information Processing Standard FIPS 186 (Digital Signature Standard, DSS), ANSI X9.30

SEE ALSO

Functions: *bn*, *dh*, *err*, *rand*, *rsa*, *sha*, *DSA_new*, *DSA_size*, *DSA_generate_parameters*, *DSA_dup_DH*, *DSA_generate_key*, *DSA_sign*, *DSA_set_method*, *DSA_get_ex_new_index*, *RSA_print*

DSA_do_sign

NAME

DSA_do_sign, DSA_do_verify – Raw DSA signature operations

SYNOPSIS

```
#include <openssl/dsa.h>
DSA_SIG *DSA_do_sign(
        const unsigned char *dgst, int dlen, DSA *dsa
);
int DSA_do_verify(
        const unsigned char *dgst, int dgst_len, DSA_SIG *sig, DSA *dsa
);
```

DESCRIPTION

The `DSA_do_sign()` function computes a digital signature on the `len` byte message digest `dgst` using the private key `dsa` and returns it in a newly allocated `DSA_SIG` structure.

The `DSA_sign_setup()` function can be used to precompute part of the signing operation in case signature generation is time-critical.

The `DSA_do_verify()` function verifies that the signature `sig` matches a given message digest `dgst` of size `len`. The `dsa` is the signer's public key.

RETURN VALUES

The `DSA_do_sign()` function returns the signature, NULL on error. The `DSA_do_verify()` function returns 1 for a valid signature, 0 for an incorrect signature and -1 on error. The error codes can be obtained from `ERR_get_error()`.

HISTORY

The `DSA_do_sign()` and `DSA_do_verify()` functions were added in OpenSSL 0.9.3.

SEE ALSO

Functions: *dsa*, *err*, *rand*, *DSA_SIG_new*, *DSA_sign*

DSA_dup_DH

NAME

DSA_dup_DH – Create a DH structure out of DSA structure

SYNOPSIS

```
#include <openssl/dsa.h>

DH * DSA_dup_DH(
        DSA *r
);
```

DESCRIPTION

The DSA_dup_DH() function duplicates DSA parameters/keys as DH parameters/keys. The q is lost during that conversion, but the resulting DH parameters contain its length.

NOTES

Be careful to avoid small subgroup attacks when using this.

RETURN VALUE

The DSA_dup_DH() function returns the new DH structure, and NULL on error. The error codes can be obtained from ERR_get_error().

HISTORY

The DSA_dup_DH() function was added in OpenSSL 0.9.4.

SEE ALSO

Functions: *dh*, *dsa*, *err*

DSA_generate_key

NAME

DSA_generate_key – Generate DSA key pair

SYNOPSIS

```
#include <openssl/dsa.h>
int DSA_generate_key(
       DSA *a
);
```

DESCRIPTION

The `DSA_generate_key()` function expects `a` to contain DSA parameters. It generates a new key pair and stores it in `a->pub_key` and `a->priv_key`.

The PRNG must be seeded prior to calling `DSA_generate_key()`.

RETURN VALUE

The `DSA_generate_key()` function returns 1 on success, 0 otherwise. The error codes can be obtained from `ERR_get_error()`.

HISTORY

The `DSA_generate_key()` function is available since SSLeay 0.8.

SEE ALSO

Functions: *dsa*, *err*, *rand*, *DSA_generate_parameters*

DSA_generate_parameters

NAME

DSA_generate_parameters – Generate DSA parameters

SYNOPSIS

```
#include <openssl/dsa.h>

DSA *DSA_generate_parameters(

        int bits, unsigned char *seed, int seed_len, int *counter_ret, unsigned long
       *h_ret, void (*callback)(int, int, void *), void *cb_arg

);
```

DESCRIPTION

The `DSA_generate_parameters()` function generates primes `p` and `q` and a generator `g` for use in the DSA.

The value of `bits` is the length of the prime to be generated; the DSS allows a maximum of 1024 bits.

If `seed` is `NULL` or `seed_len` < 20, the primes will be generated at random. Otherwise, the seed is used to generate them. If the given seed does not yield a prime `q`, a new random seed is chosen and placed at `seed`.

The `DSA_generate_parameters()` function places the iteration count in *`counter_ret` and a counter used for finding a generator in *`h_ret`, unless these are `NULL`.

A callback function may be used to provide feedback about the progress of the key generation. If `callback` is not `NULL`, it will be called as follows:

- When a candidate for `q` is generated, `callback(0, m++, cb_arg)` is called (m is 0 for the first candidate).
- When a candidate for `q` has passed a test by trial division, `callback(1, -1, cb_arg)` is called. While a candidate for `q` is tested by Miller-Rabin primality tests, `callback(1, i, cb_arg)` is called in the outer loop (once for each witness that confirms that the candidate may be prime); `i` is the loop counter (starting at 0).
- When a prime `q` has been found, `callback(2, 0, cb_arg)` and `callback(3, 0, cb_arg)` are called.
- Before a candidate for `p` (other than the first) is generated and tested, `callback(0, counter, cb_arg)` is called.
- When a candidate for `p` has passed the test by trial division, `callback(1, -1, cb_arg)` is called. While it is tested by the Miller-Rabin primality test, `callback(1, i, cb_arg)` is called in the outer loop (once for each witness that confirms that the candidate may be prime); `i` is the loop counter (starting at 0).
- When `p` has been found, `callback(2, 1, cb_arg)` is called.
- When the generator has been found, `callback(3, 1, cb_arg)` is called.

RESTRICTIONS

Seed lengths > 20 are not supported.

RETURN VALUE

The `DSA_generate_parameters()` function returns a pointer to the DSA structure, or `NULL` if the parameter generation fails. The error codes can be obtained from `ERR_get_error()`.

HISTORY

The `DSA_generate_parameters()` function appeared in SSLeay 0.8. The `cb_arg` argument was added in SSLeay 0.9.0. In versions up to OpenSSL 0.9.4, `callback(1, ...)` was called in the inner loop of the Miller-Rabin test whenever it reached the squaring step (the parameters to `callback` did not reveal how many witnesses had been tested); since OpenSSL 0.9.5, `callback(1, ...)` is called as in *BN_is_prime*, meaning once for each witness.

SEE ALSO

Functions: *dsa*, *err*, *rand*, *DSA_free*

DSA_get_ex_new_index

NAME

DSA_get_ex_new_index, DSA_set_ex_data, DSA_get_ex_data – Add application specific data to DSA structures

SYNOPSIS

```
#include <openssl/DSA.h>

int DSA_get_ex_new_index(

         long argl, void *argp, CRYPTO_EX_new *new_func, CRYPTO_EX_dup *dup_func,
        CRYPTO_EX_free *free_func

);

int DSA_set_ex_data(

        DSA *d, int idx, void *arg

);

char *DSA_get_ex_data(

        DSA *d, int idx

);
```

DESCRIPTION

These functions handle application specific data in DSA structures. Their usage is identical to that of `RSA_get_ex_new_index()`, `RSA_set_ex_data()`, and `RSA_get_ex_data()`, as described in *RSA_get_ex_new_index*>.

HISTORY

The `DSA_get_ex_new_index()`, `DSA_set_ex_data()`, and `DSA_get_ex_data()` functions are available since OpenSSL 0.9.5.

SEE ALSO

Functions: *RSA_get_ex_new_index*, *dsa*

DSA_new

NAME

DSA_new, DSA_free – Allocate and free DSA objects

SYNOPSIS

```
#include <openssl/dsa.h>
DSA* DSA_new(
        void
);
void DSA_free(
        DSA *dsa
);
```

DESCRIPTION

The `DSA_new()` function allocates and initializes a `DSA` structure.

The `DSA_free()` function frees the `DSA` structure and its components. The values are erased before the memory is returned to the system.

RETURN VALUES

If the allocation fails, the `DSA_new()` function returns `NULL` and sets an error code that can be obtained from `ERR_get_error()`. Otherwise it returns a pointer to the newly allocated structure.

The `DSA_free()` function returns no value.

HISTORY

The `DSA_new()` and `DSA_free()` functions are available in all versions of SSLeay and OpenSSL.

SEE ALSO

Functions: *dsa*, *err*, *DSA_generate_parameters*, *DSA_generate_key*

DSA_set_default_openssl_method

NAME

DSA_set_method, DSA_set_default_openssl_method, DSA_get_default_openssl_method, DSA_new_method, DSA_OpenSSL – Select DSA method

SYNOPSIS

```
#include <openssl/dsa.h>
#include <openssl/engine.h>

void DSA_set_default_openssl_method(
        DSA_METHOD *meth
);

DSA_METHOD *DSA_get_default_openssl_method(
        void
);

 int DSA_set_method(
        DSA *dsa, ENGINE *engine
);

DSA *DSA_new_method(
        ENGINE *engine
);

DSA_METHOD *DSA_OpenSSL(
         void
);
```

DESCRIPTION

A DSA_METHOD specifies the functions that OpenSSL uses for DSA operations. By modifying the method, alternative implementations such as hardware accelerators can be used.

Initially, the default is the OpenSSL internal implementation. The DSA_OpenSSL() function returns a pointer to that method.

The DSA_set_default_openssl_method() function makes meth the default method for all DSA structures created later. This is true only while the default engine for DSA operations remains as openssl. ENGINEs provide an encapsulation for implementations of one or more algorithms at a time, and all the DSA functions mentioned here operate within the scope of the default openssl engine.

The DSA_get_default_openssl_method() function returns a pointer to the current default method for the openssl engine.

The DSA_set_method() selects engine for all operations using the structure dsa.

The DSA_new_method() function allocates and initializes a DSA structure so that engine will be used for the DSA operations. If engine is NULL, the default engine for DSA operations is used.

THE DSA_METHOD STRUCTURE

```
struct { /* name of the implementation */ const char *name;
```

```
     /* sign */
DSA_SIG *(*dsa_do_sign)(const unsigned char *dgst, int dlen,
                                 DSA *dsa);

     /* pre-compute k^-1 and r */

int (*dsa_sign_setup)(DSA *dsa, BN_CTX *ctx_in, BIGNUM **kinvp,
                                 BIGNUM **rp);
     /* verify */

int (*dsa_do_verify)(const unsigned char *dgst, int dgst_len,
                                 DSA_SIG *sig, DSA *dsa);

     /* compute rr = a1^p1 * a2^p2 mod m (May be NULL for some
                                          implementations) */

int (*dsa_mod_exp)(DSA *dsa, BIGNUM *rr, BIGNUM *a1, BIGNUM *p1,
                                 BIGNUM *a2, BIGNUM *p2, BIGNUM *m,
                                 BN_CTX *ctx, BN_MONT_CTX *in_mont);

     /* compute r = a ^ p mod m (May be NULL for some implementations) */
        int (*bn_mod_exp)(DSA *dsa, BIGNUM *r, BIGNUM *a,
                                 const BIGNUM *p, const BIGNUM *m,
                                 BN_CTX *ctx, BN_MONT_CTX *m_ctx);

     /* called at DSA_new */
        int (*init)(DSA *DSA);

     /* called at DSA_free */
        int (*finish)(DSA *DSA);
        int flags;
        char *app_data; /* ?? */

 } DSA_METHOD;
```

RETURN VALUES

The DSA_OpenSSL() and DSA_get_default_openssl_method() functions return pointers to the respective DSA_METHODs.

The DSA_set_default_openssl_method() function returns no value.

The DSA_set_method() function returns non-zero if the ENGINE associated with dsa was successfully changed to engine.

The DSA_new_method() function returns NULL and sets an error code that can be obtained from ERR_get_error() if the allocation fails. Otherwise it returns a pointer to the newly allocated structure.

HISTORY

The DSA_set_default_method(), DSA_get_default_method(), DSA_set_method(), DSA_new_method(), and DSA_OpenSSL() functions were added in OpenSSL 0.9.4.

The DSA_set_default_openssl_method() and DSA_get_default_openssl_method() functions replaced the DSA_set_default_method() and DSA_get_default_method() functions respectively, and the DSA_set_method() and DSA_new_method() functions were altered to use ENGINEs rather than DSA_METHODs during development of OpenSSL 0.9.6.

SEE ALSO

Functions: *dsa*, *DSA_new*

DSA_SIG_new

NAME

DSA_SIG_new, DSA_SIG_free – Allocate and free DSA signature objects

SYNOPSIS

```
#include <openssl/dsa.h>
DSA_SIG *DSA_SIG_new(
        void
);
void DSA_SIG_free(
        DSA_SIG *a
);
```

DESCRIPTION

The `DSA_SIG_new()` function allocates and initializes a `DSA_SIG` structure.

The `DSA_SIG_free()` function frees the `DSA_SIG` structure and its components. The values are erased before the memory is returned to the system.

RETURN VALUES

If the allocation fails, `DSA_SIG_new()` returns `NULL` and sets an error code that can be obtained from `ERR_get_error()`. Otherwise it returns a pointer to the newly allocated structure.

The `DSA_SIG_free()` function returns no value.

HISTORY

The `DSA_SIG_new()` and `DSA_SIG_free()` functions were added in OpenSSL 0.9.3.

SEE ALSO

Functions: *dsa*, *err*, *DSA_do_sign*

DSA_sign

NAME

DSA_sign, DSA_sign_setup, DSA_verify – DSA signatures

SYNOPSIS

```
#include <openssl/dsa.h>

int DSA_sign(
        int type, const unsigned char *dgst, int len, unsigned char *sigret, unsigned int
        *siglen, DSA *dsa
);

int DSA_sign_setup(
        DSA *dsa, BN_CTX *ctx, BIGNUM **kinvp, BIGNUM **rp
);

int DSA_verify(
        int type, const unsigned char *dgst, int len, unsigned char *sigbuf, int siglen,
        DSA *dsa
);
```

DESCRIPTION

The `DSA_sign()` function computes a digital signature on the `len` byte message digest (`dgst`) using the private key `dsa` and places its ASN.1 DER encoding at `sigret`. The length of the signature is placed in *`siglen`. The `sigret` must point to DSA_size(`dsa`) bytes of memory.

The `DSA_sign_setup()` function may be used to precompute part of the signing operation in case signature generation is time-critical. It expects `dsa` to contain DSA parameters. It places the precomputed values in newly allocated `BIGNUM`s at *`kinvp` and *`rp`, after freeing the old ones unless *`kinvp` and *`rp` are NULL. These values may be passed to `DSA_sign()` in `dsa->kinv` and `dsa->r`. The `ctx` is a pre-allocated `BN_CTX` or NULL.

The `DSA_verify()` function verifies that the signature `sigbuf` of size `siglen` matches a given message digest `dgst` of size `len`. The `dsa` is the signer's public key.

The `type` parameter is ignored.

The PRNG must be seeded before the `DSA_sign()` or `DSA_sign_setup()` function is called.

These functions conform to US Federal Information Processing Standard FIPS 186 (Digital Signature Standard, DSS), ANSI X9.30.

RETURN VALUES

The `DSA_sign()` and `DSA_sign_setup()` functions return 1 on success, 0 on error. The `DSA_verify()` function returns 1 for a valid signature, 0 for an incorrect signature, and -1 on error. The error codes can be obtained from `ERR_get_error()`.

HISTORY

The `DSA_sign()` and `DSA_verify()` functions are available in all versions of SSLeay. The `DSA_sign_setup()` function was added in SSLeay 0.8.

DSA_size

NAME

DSA_size – Get DSA signature size

SYNOPSIS

```
#include <openssl/dsa.h>
int DSA_size(
        DSA *dsa
);
```

DESCRIPTION

The `DSA_size()` function returns the size of an ASN.1 encoded DSA signature in bytes. It can be used to determine how much memory must be allocated for a DSA signature.

The `dsa->q` must not be `NULL`.

RETURN VALUE

The size in bytes.

HISTORY

The `DSA_size()` function is available in all versions of SSLeay and OpenSSL.

SEE ALSO

Functions: *dsa*, *DSA_sign*

dsaparam

NAME

dsaparam – DSA parameter manipulation and generation

SYNOPSIS

`openssl dsaparam [-inform DER|PEM] [-outform DER|PEM] [-in` *`filename`*`] [-out` *`filename`*`] [-noout] [-text] [-C] [-rand` *`filename`*`] [-genkey] [-`*`numbits`*`]`

OPTIONS

`inform DER|PEM`

Specifies the input format. The `DER` option uses an ASN1 DER encoded form compatible with RFC2459 (PKIX) DSS-Parms that is a SEQUENCE consisting of p, q and g respectively. The PEM form is the default format. It consists of the `DER` format base64 encoded with additional header and footer lines.

`outform DER|PEM`

Specifies the output format, the options have the same meaning as the `inform` option.

`in` *`filename`*

Specifies the input filename to read parameters from or standard input if this option is not specified. If the `numbits` parameter is included then this option will be ignored.

`out` *`filename`*

Specifies the output filename parameters to. Standard output is used if this option is not present. The output filename should not be the same as the input filename.

`noout`

Inhibits the output of the encoded version of the parameters.

`text`

Prints out the DSA parameters in human readable form.

`C`

Converts the parameters into C code. The parameters can then be loaded by calling the `get_dsaXXX()` function.

`genkey`

Generates a DSA either using the specified or generated parameters.

`rand` *`file(s)`*

A file or files containing random data used to seed the random number generator, or an EGD socket. (See *RAND_egd*.) Multiple files can be specified separated by an OS-dependent character. The separator is a semicolon (;) for MS-Windows, a comma (,) for OpenVMS, and a colon (:) for all others.

`numbits`

Specifies that a parameter set should be generated of size `numbits`. It must be the last option. If this option is included then the input file (if any) is ignored.

DESCRIPTION

This command is used to manipulate or generate DSA parameter files.

NOTES

PEM format DSA parameters use the following header and footer lines:

```
-----BEGIN DSA PARAMETERS-----
-----END DSA PARAMETERS-----
```

DSA parameter generation is a slow process, and as a result the same set of DSA parameters is often used to generate several distinct keys.

SEE ALSO

Commands: *gendsa*, *dsa*, *genrsa*, *rsa*

enc

NAME

enc – Symmetric cipher routines

SYNOPSIS

openssl enc [-ciphername] [-in *filename*] [-out *filename*] [-pass *arg*] [-salt] [-nosalt] [-e] [-d] [-a] [-A] [-k *password*] [-kfile *filename*] [-S salt] [-K key] [-ivIV] [-p] [-P] [-bufsize *number*] [-debug]

OPTIONS

in *filename*

Input filename, standard input by default.

out *filename*

Output filename, standard output by default.

pass *arg*

Password source. For more information about the format of arg see the Pass Phrase Arguments section in *openssl*.

salt

Uses a salt in the key derivation routines. This option should always be used unless compatibility with previous versions of OpenSSL or SSLeay is required. This option is only present on OpenSSL versions 0.9.5 or above.

nosalt

Does not use a salt in the key derivation routines. This is the default for compatibility with previous versions of OpenSSL and SSLeay.

e

Encrypts the input data. This is the default.

d

Decrypts the input data.

a

Base64 processes the data. This means that if encryption is taking place the data is base64 encoded after encryption. If decryption is set then the input data is base64 decoded before being decrypted.

A

If the a option is set then base64 processes the data on one line.

k*password*

The password to derive the key from. This is for compatibility with previous versions of OpenSSL. It is superseded by the pass argument.

kfile *filename*

Reads the password to derive the key from the first line of *filename*. This is for computability with previous versions of OpenSSL. It is superseded by the `pass` argument.

`S salt`

The actual salt to use. This must be represented as a string comprised only of hex digits.

`K key`

The actual key to use. This must be represented as a string comprised only of hex digits. If only the key is specified, the IV must also be specifed using the `iv` option. When both a key and password are specified, the key given with the `K` option will be used, and the IV generated from the password will be taken. It probably does not make much sense to specify both key and password.

`ivIV`

The actual IV to use. This must be represented as a string comprised only of hex digits. When only the key is specified using the `K` option, the IV must explicitely be defined. When a password is specified using one of the other options, the IV is generated from this password.

`p`

Prints out the key and IV used.

`P`

Prints out the key and IV used then immediately exits. Does not do any encryption or decryption.

`bufsize` *number*

Sets the buffer size for I/O

`debug`

Debugs the BIOs used for I/O.

DESCRIPTION

The symmetric cipher commands allow data to be encrypted or decrypted using various block and stream ciphers using keys based on passwords or explicitly provided. Base64 encoding or decoding can also be performed either by itself or in addition to the encryption or decryption.

NOTES

The program can be called either as `openssl ciphername` or `openssl enc ciphername`.

There is a prompt for a password to derive the key and IV if necessary.

The `salt` option should always be used if the key is being derived from a password unless you want compatibility with previous versions of OpenSSL and SSLeay.

Without the `salt` option it is possible to perform efficient dictionary attacks on the password and to attack stream cipher encrypted data. The reason for this is that without the salt the same password always generates the same encryption key. When the salt is being used the first eight bytes of the encrypted data are reserved for the salt. It is generated at random when encrypting a file and read from the encrypted file when it is decrypted.

Some of the ciphers do not have large keys and others have security implications if not used correctly. A beginner is advised to use a strong block cipher in CBC mode such as `bf` or `des3`.

All the block ciphers use PKCS#5 padding, also known as standard block padding. This allows a rudimentary integrity or password check to be performed. However, since the chance of random data passing the test is better than 1 in 256 it is not a very good test.

All RC2 ciphers have the same key and effective key length.

Blowfish and RC5 algorithms use a 128 bit key.

Supported Ciphers

```
base64            Base 64
bf-cbc            Blowfish in CBC mode
bf                Alias for bf-cbc
bf-cfb            Blowfish in CFB mode
bf-ecb            Blowfish in ECB mode
bf-ofb            Blowfish in OFB mode
cast-cbc          CAST in CBC mode

cast              Alias for cast-cbc
cast5-cbc         CAST5 in CBC mode
cast5-cfb         CAST5 in CFB mode
cast5-ecb         CAST5 in ECB mode
cast5-ofb         CAST5 in OFB mode

des-cbc           DES in CBC mode
des               Alias for des-cbc
des-cfb           DES in CBC mode
des-ofb           DES in OFB mode
des-ecb           DES in ECB mode

des-ede-cbc       Two key triple DES EDE in CBC mode
des-ede           Alias for des-ede
des-ede-cfb       Two key triple DES EDE in CFB mode
des-ede-ofb       Two key triple DES EDE in OFB mode

des-ede3-cbc      Three key triple DES EDE in CBC mode
des-ede3          Alias for des-ede3-cbc
des3              Alias for des-ede3-cbc
des-ede3-cfb      Three key triple DES EDE CFB mode
des-ede3-ofb      Three key triple DES EDE in OFB mode

desx              DESX algorithm.

idea-cbc          IDEA algorithm in CBC mode
idea              same as idea-cbc
idea-cfb          IDEA in CFB mode
idea-ecb          IDEA in ECB mode
idea-ofb          IDEA in OFB mode

rc2-cbc           128 bit RC2 in CBC mode
rc2               Alias for rc2-cbc
rc2-cfb           128 bit RC2 in CBC mode
rc2-ecb           128 bit RC2 in CBC mode
rc2-ofb           128 bit RC2 in CBC mode
rc2-64-cbc        64 bit RC2 in CBC mode
rc2-40-cbc        40 bit RC2 in CBC mode

rc4               128 bit RC4
rc4-64            64 bit RC4
```

```
rc4-40            40 bit RC4

rc5-cbc           RC5 cipher in CBC mode
rc5               Alias for rc5-cbc
rc5-cfb           RC5 cipher in CBC mode
rc5-ecb           RC5 cipher in CBC mode
rc5-ofb           RC5 cipher in CBC mode
```

RESTRICTIONS

The `A` option when used with large files does not work properly.

There should be an option to allow an iteration count to be included.

Like the EVP library the `enc` program only supports a fixed number of algorithms with certain parameters. For example, if you want to use RC2 with a 76 bit key or RC4 with an 84 bit key you cannot use this program.

EXAMPLES

Just base64 encode a binary file:

```
openssl base64 -in file.bin -out file.b64
```

Decode the same file

```
openssl base64 -d -in file.b64 -out file.bin
```

Encrypt a file using triple DES in CBC mode using a prompted password:

```
openssl des3 -salt -in file.txt -out file.des3
```

Decrypt a file using a supplied password:

```
openssl des3 -d -salt -in file.des3 -out file.txt -k mypassword
```

Encrypt a file then base64 encode it (so it can be sent via mail for example) using Blowfish in CBC mode:

```
openssl bf -a -salt -in file.txt -out file.bf
```

Base64 decode a file then decrypt it:

```
openssl bf -d -salt -a -in file.bf -out file.txt
```

Decrypt some data using a supplied 40 bit RC4 key:

```
openssl rc4-40 -in file.rc4 -out file.txt -K 0102030405
```

err

NAME

err – Error codes

SYNOPSIS

```
#include <openssl/err.h>
unsigned long ERR_get_error(
        void
);
unsigned long ERR_peek_error(
        void
);
unsigned long ERR_get_error_line(
        const char **file, int *line
);
unsigned long ERR_peek_error_line(
       const char **file, int *line
);
unsigned long ERR_get_error_line_data(
       const char **file, int *line, const char **data, int *flags
);
unsigned long ERR_peek_error_line_data(
       const char **file, int *line, const char **data, int *flags
);
int ERR_GET_LIB(
       unsigned long e
);
int ERR_GET_FUNC(
       unsigned long e
);
int ERR_GET_REASON(
       unsigned long e
);
void ERR_clear_error(
       void
);
char *ERR_error_string(
       unsigned long e, char *buf
);
```

```
const char*ERR_lib_error_string(
        unsigned long e
);
const char*ERR_func_error_string(
        unsigned long e
);
const char*ERR_reason_error_string(
        unsigned long e
);
void ERR_print_errors(
        BIO *bp
);
void ERR_print_errors_fp(
        FILE *fp
);
void ERR_load_crypto_strings(
        void
);
void ERR_free_strings(
        void
);
void ERR_remove_state(
        unsigned long pid
);
void ERR_put_error(
        int lib, int func, int reason, const char *file, int line
);
void ERR_add_error_data(
        int num, ...
);
void ERR_load_strings(
        int lib,ERR_STRING_DATA str[]
);
unsigned long ERR_PACK(
        int lib, int func, int reason
);
int ERR_get_next_error_library(
        void
);
```

DESCRIPTION

When a call to the OpenSSL library fails, this is usually signalled by the return value, and an error code is stored in an error queue associated with the current thread. The `err` library provides functions to obtain these error codes and textual error messages.

The *ERR_get_error* reference page describes how to access error codes.

Error codes contain information about where the error occurred, and what went wrong. The *ERR_GET_LIB* reference page describes how to extract this information. A method to obtain human-readable error messages is described in *ERR_error_string*.

The `ERR_clear_error()` function can be used to clear the error queue.

The `ERR_remove_state()` function should be used to avoid memory leaks when threads are terminated.

ERRORS

See *ERR_put_error* if you want to record error codes in the OpenSSL error system from within your application.

The remainder of this section explains how to add new error codes to OpenSSL or add error codes from external libraries.

Reporting errors

Each sublibrary has a specific macro, `XXXerr()`, that is used to report errors. Its first argument is a function code, `XXX_F_...`, and the second argument is a reason code, `XXX_R_...`. Function codes are derived from the function names; reason codes consist of textual error descriptions. For example, the `ssl23_read()` function reports a handshake failure as follows:

```
SSLerr(SSL_F_SSL23_READ, SSL_R_SSL_HANDSHAKE_FAILURE);
```

Function and reason codes should consist of upper case characters, numbers and underscores only. The error file generation script translates function codes into function names by looking in the header files for an appropriate function name. If none is found it just uses the capitalized form, such as SSL23_READ in the previous example.

The trailing section of a reason code (after the _R_) is translated into lower case, and underscores are changed to spaces.

When you are using new function or reason codes, run the `make errors` command. The necessary `#define`s will automatically be added to the sublibrary's header file.

Although a library will normally report errors using its own specific `XXXerr()` macro, another library's macro can be used. This is usually done when a library wants to include ASN1 code which must use the `ASN1err()` macro.

Adding new libraries

When adding a new sublibrary to OpenSSL, take the following steps:

1. Assign it a library number, `ERR_LIB_XXX`.
2. Define a macro, `XXXerr()`, (both in `err.h`).
3. Add its name to `ERR_str_libraries[]` (in `crypto/err/err.c`).
4. Add `ERR_load_XXX_strings` to the `ERR_load_crypto_strings()` function (in `crypto/err/err_all.c`).

5. Add an entry, `LXXXxxx.hxxx_err.c`, to `crypto/err/openssl.ec`, and add `xxx_err.c` to the Makefile.

   Running `make errors` will generate a file, `xxx_err.c`, and add all error codes used in the library to `xxx.h`.

In addition, the library include file must have a certain form. Typically it will initially look like the following example:

```
#ifndef HEADER_XXX_H
#define HEADER_XXX_H

#ifdef __cplusplus
extern "C" {
#endif

/* Include files */

#include <openssl/bio.h>
#include <openssl/x509.h>

/* Macros, structures and function prototypes */
/* BEGIN ERROR CODES */
```

The `BEGIN ERROR CODES` sequence is used by the error code generation script as the point to place new error codes. Any text after this point will be overwritten when `make errors` is run. The closing `#endif` will be added automatically by the script.

The generated C error code file `xxx_err.c` will load the header files `stdio.h`, `openssl/err.h` and `openssl/xxx.h` so the header file must load any additional header files containing any definitions it uses.

Using Error Codes in External Libraries

It also is possible to use OpenSSL's error code scheme in external libraries. The library needs to load its own codes and call the OpenSSL error code insertion script `mkerr.pl` explicitly to add codes to the header file and generate the C error code file. This will normally be done if the external library needs to generate new ASN1 structures but it can also be used to add more general purpose error code handling.

Internals

The error queues are stored in a hash table with one `ERR_STATE` entry for each pid. The `ERR_get_state()` funciton returns the current thread's `ERR_STATE`. An `ERR_STATE` can hold up to `ERR_NUM_ERRORS` error codes. When more error codes are added, the old ones are overwritten, on the assumption that the most recent errors are most important.

Error strings are also stored in hash tables. The hash tables can be obtained by calling `ERR_get_err_state_table(void)` and `ERR_get_string_table(void)`, respectively.

SEE ALSO

Functions: *CRYPTO_set_id_callback*, *CRYPTO_set_locking_callback*, *ERR_get_error*, *ERR_GET_LIB*, *ERR_clear_error*, *ERR_error_string*, *ERR_print_errors*, *ERR_load_crypto_strings*, *ERR_remove_state*, *ERR_put_error*, *ERR_load_strings*, *SSL_get_error*

ERR_clear_error

NAME

ERR_clear_error – Clear the error queue

SYNOPSIS

```
#include <openssl/err.h>
void ERR_clear_error(
        void
);
```

DESCRIPTION

The `ERR_clear_error()` function empties the current thread's error queue.

RETURN VALUES

The `ERR_clear_error()` function has no return value.

HISTORY

The `ERR_clear_error()` function is available in all versions of SSLeay and OpenSSL.

SEE ALSO

Functions: *err*, *ERR_get_error*

ERR_error_string

NAME

ERR_error_string, ERR_error_string_n, ERR_lib_error_string, ERR_func_error_string, ERR_reason_error_string – Obtain human-readable error message

SYNOPSIS

```
#include <openssl/err.h>
char *ERR_error_string(
       unsigned long e, char *buf
);
char *ERR_error_string_n(
        unsigned long e, char *buf, size_t len
);
const char *ERR_lib_error_string(
        unsigned long e
);
const char *ERR_func_error_string(
       unsigned long e
);
const char *ERR_reason_error_string(
       unsigned long e
);
```

DESCRIPTION

The `ERR_error_string()` function generates a human-readable string representing the error code *e*, and places it at *buf*. The *buf* must be at least 120 bytes long. If *buf* is `NULL`, the error string is placed in a static buffer. The `ERR_error_string_n()` function is a variant of the `ERR_error_string()` functon that writes at most *len* characters (including the terminating 0) and truncates the string if necessary. For `ERR_error_string_n()`, *buf* cannot be `NULL`.

The string will have the following format:

```
error:[error code]:[library name]:[function name]:[reason string]
```

The *error code* is an 8-digit hexadecimal number. The *library name*, *function name*, and *reason string* are ASCII text.

The `ERR_lib_error_string()`, `ERR_func_error_string()`, and `ERR_reason_error_string()` functions return the library name, function name and reason string respectively.

The OpenSSL error strings should be loaded by first calling `ERR_load_crypto_strings()` or, for SSL applications, `SSL_load_error_strings()`. If there is no text string registered for the given error code, the error string will contain the numeric code.

The `ERR_print_errors()` function can be used to print all error codes currently in the queue.

RETURN VALUES

The `ERR_error_string()` function returns a pointer to a static buffer containing the string if *buf* == `NULL`, *buf* otherwise.

The `ERR_lib_error_string()`, `ERR_func_error_string()`, and `ERR_reason_error_string()` functions return the strings, and `NULL` if none is registered for the error code.

HISTORY

The `ERR_error_string()` function is available in all versions of SSLeay and OpenSSL. The `ERR_error_string_n()` function was added in OpenSSL 0.9.6.

SEE ALSO

Functions: *err*, *ERR_get_error*, *ERR_load_crypto_strings*, *SSL_load_error_strings* *ERR_print_errors*

ERR_get_error

NAME

ERR_get_error, ERR_peek_error, ERR_get_error_line, ERR_peek_error_line, ERR_get_error_line_data, ERR_peek_error_line_data – Obtain error code and data

SYNOPSIS

```
#include <openssl/err.h>
unsigned long ERR_get_error(
        void
);
unsigned long ERR_peek_error(
        void
);
unsigned long ERR_get_error_line(
        unsigned long const char **file, int *line
);
unsigned long ERR_peek_error_line(
       const char **file, int *line
);
unsigned long ERR_get_error_line_data(
       const char **file, int *line, const char **data, int *flags
);
unsigned long ERR_peek_error_line_data(
       const char **file, int *line, const char **data, int *flags
);
```

DESCRIPTION

The `ERR_get_error()` function returns the last error code from the thread's error queue and removes the entry. This function can be called repeatedly until there are no more error codes to return.

The `ERR_peek_error()` function returns the last error code from the thread's error queue without modifying it.

See *ERR_GET_LIB* for information about location and reason of the error, and *ERR_error_string* for human-readable error messages.

The `ERR_get_error_line()` and `ERR_peek_error_line()` functions are the same as the above, but they also store the file name and line number where the error occurred in *`file` and *`line`, unless these are `NULL`.

The `ERR_get_error_line_data()` and `ERR_peek_error_line_data()` functions store additional data and flags associated with the error code in *`data` and *`flags`, unless these are `NULL`. The *`data` contains a string if *`flags`&`ERR_TXT_STRING`. If it has been allocated by `OPENSSL_malloc()`, *`flags`&`ERR_TXT_MALLOCED` is true.

RETURN VALUES

The error code, or 0 if there is no error in the queue.

HISTORY

The `ERR_get_error()`, `ERR_peek_error()`, `ERR_get_error_line()`, and `ERR_peek_error_line()` functions are available in all versions of SSLeay and OpenSSL. The `ERR_get_error_line_data()` and `ERR_peek_error_line_data()` functions were added in SSLeay 0.9.0.

SEE ALSO

Functions: *err*, *ERR_error_string*, *ERR_GET_LIB*

ERR_GET_LIB

NAME

ERR_GET_LIB, ERR_GET_FUNC, ERR_GET_REASON – Get library, function and reason code

SYNOPSIS

```
#include <openssl/err.h>
int ERR_GET_LIB(
       unsigned long e
);
int ERR_GET_FUNC(
       unsigned long e
);
int ERR_GET_REASON(
       unsigned long e
);
```

DESCRIPTION

The error code returned by the `ERR_get_error()` function consists of a library number, function code and reason code. The `ERR_GET_LIB()`, `ERR_GET_FUNC()`, and `ERR_GET_REASON()` macros can be used to extract these.

The library number and function code describe where the error occurred. The reason code is the information about what went wrong.

Each sublibrary of OpenSSL has a unique library number; function and reason codes are unique within each sublibrary. Different libraries may use the same value to signal different functions and reasons.

The `ERR_R_...` reason codes, such as `ERR_R_MALLOC_FAILURE`, are globally unique. However, when checking for sublibrary specific reason codes, be sure to compare the library number.

RETURN VALUES

These functions return the library number, function code, and reason code respectively.

HISTORY

`ERR_GET_LIB()`, `ERR_GET_FUNC()`, and `ERR_GET_REASON()` are available in all versions of SSLeay and OpenSSL.

SEE ALSO

Functions: *err*, *ERR_get_error*

ERR_load_crypto_strings

NAME

ERR_load_crypto_strings, SSL_load_error_strings, ERR_free_strings – Load and free error strings

SYNOPSIS

```
#include <openssl/err.h>
void ERR_load_crypto_strings(
        void
);
void ERR_free_strings(
       void
);
#include <openssl/ssl.h>
void SSL_load_error_strings(
        void
);
```

DESCRIPTION

The `ERR_load_crypto_strings()` function registers the error strings for all `libcrypto` functions. The `SSL_load_error_strings()` function does the same, but also registers the `libssl` error strings.

One of these functions should be called before generating textual error messages. However, this is not required when memory usage is an issue.

The `ERR_free_strings()` function frees all previously loaded error strings.

RETURN VALUES

The `ERR_load_crypto_strings()`, `SSL_load_error_strings()`, and `ERR_free_strings()` functions return no values.

HISTORY

The `ERR_load_error_strings()`, `SSL_load_error_strings()`, and `ERR_free_strings()` functions are available in all versions of SSLeay and OpenSSL.

SEE ALSO

Functions: *err*, *ERR_error_string*

ERR_load_SSL_strings

NAME

ERR_load_SSL_strings, ERR_load_crypto_strings, SSL_load_error_strings – Load error strings

SYNOPSIS

```
#include <openssl/err.h>

        void ERR_load_crypto_strings(

void

);
       void ERR_free_strings(

void

);

#include <openssl/ssl.h>

        void SSL_load_error_strings(

void

);

#include <openssl/err.h>
#include <openssl/ssl.h>

        void ERR_load_SSL_strings(

void

);
```

DESCRIPTION

The `ERR_load_crypto_strings()` function registers the error strings for all `libcrypto` functions. The `ERR_load_SSL_strings()` function registers the error strings for all `libssl` functions. The `SSL_load_error_strings()` function registers both of the `libcrypto` and `libssl` error strings.

One of these functions should be called before generating textual error messages. However, this is not required when memory usage is an issue.

The `ERR_free_strings()` function frees all previously loaded error strings.

RETURN VALUES

The `ERR_load_crypto_strings()`, `ERR_load_SSL_strings()`, `SSL_load_error_strings`, and `ERR_free_strings()` functions return no values.

SEE ALSO

Functions: *err*, *ERR_error_string*

ERR_load_strings

NAME

ERR_load_strings, ERR_PACK, ERR_get_next_error_library – Load arbitrary error strings

SYNOPSIS

```
#include <openssl/err.h>

void ERR_load_strings(
        int lib, ERR_STRING_DATA str[]
);

int ERR_get_next_error_library(
         void
);

unsigned long ERR_PACK(
        int lib, int func, int reason
);
```

DESCRIPTION

The `ERR_load_strings()` function registers error strings for library number `lib`.

The `str` is an array of error string data:

```
typedef struct ERR_string_data_st
{
       unsigned long error;
       char *string;
} ERR_STRING_DATA;
```

The error code is generated from the library number and a function and reason code: `error = ERR_PACK(lib, func, reason)`. `ERR_PACK()` is a macro.

The last entry in the array is {0,0}.

The `ERR_get_next_error_library()` function can be used to assign library numbers to user libraries at runtime.

RETURN VALUE

The `ERR_load_strings()` function returns no value. The `ERR_PACK()` function returns the error code. The `ERR_get_next_error_library()` function returns a new library number.

HISTORY

The `ERR_load_error_strings()` and `ERR_PACK()` functions are available in all versions of SSLeay and OpenSSL. The `ERR_get_next_error_library()` function was added in SSLeay 0.9.0.

SEE ALSO

Functions: *err*, *ERR_load_strings*

ERR_print_errors

NAME

ERR_print_errors, ERR_print_errors_fp – Print error messages

SYNOPSIS

```
#include <openssl/err.h>
void ERR_print_errors(
       BIO *bp
);
void ERR_print_errors_fp(
        FILE *fp
);
```

DESCRIPTION

`ERR_print_errors()` is a convenience function that prints the error strings for all errors that OpenSSL has recorded to `bp`, thus emptying the error queue.

The `ERR_print_errors_fp()` function is the same, except that the output goes to a `FILE`.

The error strings will have the following format:

```
[pid]:error:[error code]:[library name]:[function name]:[reason string]
:[file name]:[line]:[optional text message]
```

The *`error code`* is an 8-digit hexadecimal number. The *`library name`*, *`function name`* and *`reason string`* are ASCII text, as is *`optional text message`* if one was set for the respective error code.

If there is no text string registered for the given error code, the error string will contain the numeric code.

RETURN VALUES

The `ERR_print_errors()` and `ERR_print_errors_fp()` functions return no values.

HISTORY

The `ERR_print_errors()` and `ERR_print_errors_fp()` functions are available in all versions of SSLeay and OpenSSL.

SEE ALSO

Functions: *err*, *ERR_error_string*, *ERR_get_error*, *ERR_load_crypto_strings*, *SSL_load_error_strings*

ERR_put_error

NAME

ERR_put_error, ERR_add_error_data – Record an error

SYNOPSIS

```
#include <openssl/err.h>
void ERR_put_error(
       int lib, int func, int reason, const char *file, int line
);
void ERR_add_error_data(
        int num, ...
);
```

DESCRIPTION

The `ERR_put_error()` function adds an error code to the thread's error queue. It signals that the error of reason code `reason` occurred in function `func` of library `lib`, in line number `line` of `file`. This function is usually called by a macro.

The `ERR_add_error_data()` function associates the concatenation of its `num` string arguments with the error code added last.

The `ERR_load_strings()` function can be used to register error strings so that the application can generate human-readable error messages for the error code.

RETURN VALUES

The `ERR_put_error()` and `ERR_add_error_data()` functions return no values.

HISTORY

The `ERR_put_error()` function is available in all versions of SSLeay and OpenSSL. The `ERR_add_error_data()` function was added in SSLeay 0.9.0.

SEE ALSO

Functions: *err*, *ERR_load_strings*

ERR_remove_state

NAME

ERR_remove_state – Free a thread's error queue

SYNOPSIS

```
#include <openssl/err.h>
void ERR_remove_state(
       unsigned long pid
);
```

DESCRIPTION

The `ERR_remove_state()` function frees the error queue associated with thread `pid`. If `pid` == 0, the current thread will have its error queue removed.

Since error queue data structures are allocated automatically for new threads, they must be freed when threads are terminated in order to avoid memory leaks.

RETURN VALUE

The `ERR_remove_state()` function returns no value.

HISTORY

The `ERR_remove_state()` function is available in all versions of SSLeay and OpenSSL.

SEE ALSO

Functions: *err*

evp

NAME

evp – High-level cryptographic functions

SYNOPSIS

#include <openssl/evp.h>

DESCRIPTION

The EVP library provides a high-level interface to cryptographic functions.

`EVP_Seal...` and `EVP_Open...` provide public key encryption and decryption to implement digital envelopes.

The `EVP_Sign...` and `EVP_Verify...` functions implement digital signatures.

Symmetric encryption is available with the `EVP_Encrypt...` functions. The `EVP_Digest...` functions provide message digests.

Algorithms are loaded with `OpenSSL_add_all_algorithms()`.

SEE ALSO

Functions: *EVP_DigestInit*, *EVP_EncryptInit*, *EVP_OpenInit*, *EVP_SealInit*, *EVP_SignInit*, *EVP_VerifyInit*, *OpenSSL_add_all_algorithms*

EVP_DigestInit

NAME

EVP_DigestInit, EVP_DigestUpdate, EVP_DigestFinal, EVP_MAX_MD_SIZE, EVP_MD_CTX_copy, EVP_MD_type, EVP_MD_pkey_type, EVP_MD_size, EVP_MD_block_size, EVP_MD_CTX_md, EVP_MD_CTX_size, EVP_MD_CTX_block_size, EVP_MD_CTX_type, EVP_md_null, EVP_md2, EVP_md5EVP_sha, EVP_sha1, EVP_dss, EVP_dss1, EVP_mdc2, EVP_ripemd160, EVP_get_digestbyname, EVP_get_digestbynid, EVP_get_digestbyobj – EVP digest routines

SYNOPSIS

```
#include <openssl/evp.h>

void EVP_DigestInit(
        EVP_MD_CTX *ctx, const EVP_MD *type
);

void EVP_DigestUpdate(
        EVP_MD_CTX *ctx, const void *d, unsigned int cnt
);

void EVP_DigestFinal(
        EVP_MD_CTX *ctx, unsigned char *md, unsigned int *s
);

#define EVP_MAX_MD_SIZE (16+20) /* The SSLv3 md5+sha1 type */

int EVP_MD_CTX_copy(
        EVP_MD_CTX *out,EVP_MD_CTX *in
);

#define EVP_MD_type(e)((e)->type)
#define EVP_MD_pkey_type(e)((e)->pkey_type)
#define EVP_MD_size(e)((e)->md_size)
#define EVP_MD_block_size(e)((e)->block_size)
#define EVP_MD_CTX_md(e)(e)->digest)
#define EVP_MD_CTX_size(e)EVP_MD_size((e)->digest)
#define EVP_MD_CTX_block_size(e)EVP_MD_block_size((e)->digest)
#define EVP_MD_CTX_type(e)EVP_MD_type((e)->digest)

EVP_MD *EVP_md_null(
        void
);

EVP_MD *EVP_md2(
        void
);

EVP_MD *EVP_md5(
        void
);

EVP_MD *EVP_sha(
        void
);
```

```
EVP_MD *EVP_sha1(
        void
);
EVP_MD *EVP_dss(
        void
);
EVP_MD *EVP_dss1(
        void
);
EVP_MD *EVP_mdc2(
        void
);
EVP_MD *EVP_ripemd160(
        void
);
const EVP_MD *EVP_get_digestbyname(
        const char *name
);
#define EVP_get_digestbynid(a) EVP_get_digestbyname(OBJ_nid2sn(a))
#define EVP_get_digestbyobj(a) EVP_get_digestbynid(OBJ_obj2nid(a))
```

DESCRIPTION

The EVP digest routines are a high level interface to message digests.

The `EVP_DigestInit()` function initializes a digest context `ctx` to use a digest `type`. This will typically be supplied by a function such as `EVP_sha1()`.

The `EVP_DigestUpdate()` function hashes `cnt` bytes of data at `d` into the digest context `ctx`. This function can be called several times on the same `ctx` to hash additional data.

The `EVP_DigestFinal()` function retrieves the digest value from `ctx` and places it in `md`. If the `s` parameter is not NULL then the number of bytes of data written (i.e. the length of the digest) will be written to the integer at `s`. At most `EVP_MAX_MD_SIZE` bytes will be written. After calling `EVP_DigestFinal()` no additional calls to `EVP_DigestUpdate()` can be made, but `EVP_DigestInit()` can be called to initialize a new digest operation.

The `EVP_MD_CTX_copy()` function can be used to copy the message digest state from `in` to `out`. This is useful to hash large amounts of data which only differ in the last few bytes.

The `EVP_MD_size()` and `EVP_MD_CTX_size()` functions return the size of the message digest when passed an `EVP_MD` or an `EVP_MD_CTX` structure, i.e. the size of the hash.

The `EVP_MD_block_size()` and `EVP_MD_CTX_block_size()` functions return the block size of the message digest when passed an `EVP_MD` or an `EVP_MD_CTX` structure.

The `EVP_MD_type()` and `EVP_MD_CTX_type()` functions return the NID of the OBJECT IDENTIFIER representing the given message digest when passed an `EVP_MD` structure. For example, `EVP_MD_type(EVP_sha1())` returns `NID_sha1`. This function is normally used when setting ASN1 OIDs.

The `EVP_MD_CTX_md()` function returns the `EVP_MD` structure corresponding to the passed `EVP_MD_CTX`.

The EVP_MD_pkey_type() function returns the NID of the public key signing algorithm associated with this digest. For example, EVP_sha1() is associated with RSA so this will return NID_sha1WithRSAEncryption.

The EVP_md2(), EVP_md5(), EVP_sha(), EVP_sha1(), EVP_mdc2(), and EVP_ripemd160() functions return EVP_MD structures for the MD2, MD5, SHA, SHA1, MDC2 and RIPEMD160 digest algorithms respectively. The associated signature algorithm is RSA in each case.

The EVP_dss() and EVP_dss1() functions return EVP_MD structures for SHA and SHA1 digest algorithms but using DSS (DSA) for the signature algorithm.

The EVP_md_null() function is a null message digest that does nothing. The hash it returns is of zero length.

The EVP_get_digestbyname(), EVP_get_digestbynid(), and EVP_get_digestbyobj() functions return an EVP_MD structure when passed a digest name, a digest NID or an ASN1_OBJECT structure respectively. The digest table must be initialized using, for example, OpenSSL_add_all_digests(), for these functions to work.

NOTES

The EVP interface to message digests should almost always be used in preference to the low level interfaces. This is because the code then becomes transparent to the digest used and much more flexible.

SHA1 is the digest of choice for new applications. The other digest algorithms are still in common use.

RESTRICTIONS

Several of the functions do not return values. Although the internal digest operations will never fail some future hardware based operations might.

The link between digests and signing algorithms results in a situation where the EVP_sha1() function must be used with RSA, and the EVP_dss1() function must be used with DSS even though they are identical digests.

The size of an EVP_MD_CTX structure is determined at compile time. This results in code that must be recompiled if the size of EVP_MD_CTX increases.

RETURN VALUES

The EVP_DigestInit(), EVP_DigestUpdate(), and EVP_DigestFinal() functions do not return values.

The EVP_MD_CTX_copy() function returns 1 if successful or 0 for failure.

The EVP_MD_type(), EVP_MD_pkey_type(), and EVP_MD_type() functions return the NID of the corresponding OBJECT IDENTIFIER or NID_undef if none exists.

The EVP_MD_size(), EVP_MD_block_size(), EVP_MD_CTX_size(), EVP_MD_size(), EVP_MD_CTX_block_size(), and EVP_MD_block_size() functions return the digest or block size in bytes.

The EVP_md_null(), EVP_md2(), EVP_md5(), EVP_sha(), EVP_sha1(), EVP_dss(), EVP_dss1(), EVP_mdc2(), and EVP_ripemd160() functions return pointers to the corresponding EVP_MD structures.

The EVP_get_digestbyname(), EVP_get_digestbynid(), and EVP_get_digestbyobj() functions return either an EVP_MD structure or NULL if an error occurs.

EXAMPLE

The following example digests the data "Test Message\n" and "Hello World\n", using the digest name passed on the command line:

```
#include <stdio.h>
#include <openssl/evp.h>

main(int argc, char *argv[])
{

EVP_MD_CTX mdctx;
const EVP_MD *md;
char mess1[] = "Test Message\n";
char mess2[] = "Hello World\n";
unsigned char md_value[EVP_MAX_MD_SIZE];
int md_len, i;

OpenSSL_add_all_digests();

if(!argv[1]) {
printf("Usage: mdtest digestname\n");
exit (1);
}

md = EVP_get_digestbyname(argv[1]);

if(!md) {
printf("Unknown message digest %s\n", argv[1]);
exit (1);
}

EVP_DigestInit(&mdctx, md);
EVP_DigestUpdate(&mdctx, mess1, strlen(mess1));
EVP_DigestUpdate(&mdctx, mess2, strlen(mess2));
EVP_DigestFinal(&mdctx, md_value, &md_len);

printf("Digest is: ");
for(i = 0; i < md_len; i++) printf("%02x", md_value[i]);
printf("\n");
}
```

HISTORY

`EVP_DigestInit()`, `EVP_DigestUpdate()`, and `EVP_DigestFinal()` are available in all versions of SSLeay and OpenSSL.

SEE ALSO

Commands: *digest*

Functions: *evp*, *hmac*, *md2*, *md5*, *mdc2*, *ripemd160*, *sha*

EVP_EncryptInit

NAME

EVP_EncryptInit, EVP_EncryptUpdate, EVP_EncryptFinal, EVP_DecryptInit, EVP_DecryptUpdate, EVP_DecryptFinal, EVP_CipherInit, EVP_CipherUpdate, EVP_CipherFinal, EVP_CIPHER_CTX_set_key_length, EVP_CIPHER_CTX_ctrl, EVP_CIPHER_CTX_cleanup, EVP_get_cipherbynam, EVP_get_cipherbynid, EVP_get_cipherbyobj, EVP_CIPHER_nid, EVP_CIPHER_block_size, EVP_CIPHER_key_length, EVP_CIPHER_iv_length, EVP_CIPHER_flags, EVP_CIPHER_mode, EVP_CIPHER_type, EVP_CIPHER_CTX_cipher, EVP_CIPHER_CTX_nid, EVP_CIPHER_CTX_block_size, EVP_CIPHER_CTX_key_length, EVP_CIPHER_CTX_iv_length, EVP_CIPHER_CTX_get_app_data, EVP_CIPHER_CTX_set_app_data, EVP_CIPHER_CTX_type, EVP_CIPHER_CTX_flags, EVP_CIPHER_CTX_mode, EVP_CIPHER_param_to_asn1, EVP_CIPHER_asn1_to_param – EVP cipher routines

SYNOPSIS

```
#include <openssl/evp.h>

int EVP_EncryptInit(
       EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type, unsigned char *key, unsigned char
       *iv
);

int EVP_EncryptUpdate(
       EVP_CIPHER_CTX *ctx, unsigned char *out, int *outl, unsigned char *in, int inl
);

int EVP_EncryptFinal(
       EVP_CIPHER_CTX *ctx, unsigned char *out, int *outl
);

int EVP_DecryptInit(
       EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type, unsigned char *key, unsigned char
       *iv
);

int EVP_DecryptUpdate(
       EVP_CIPHER_CTX *ctx, unsigned char *out, int *outl, unsigned char *in, int inl
);

int EVP_DecryptFinal(
       EVP_CIPHER_CTX *ctx, unsigned char *outm, int *outl
);

int EVP_CipherInit(
       EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type, unsigned char *key, unsigned char
       *iv, int enc
);

int EVP_CipherUpdate(
       EVP_CIPHER_CTX *ctx, unsigned char *out, int *outl, unsigned char *in, int inl
);
```

```
int EVP_CipherFinal(
        EVP_CIPHER_CTX *ctx, unsigned char *outm, int *outl
);
int EVP_CIPHER_CTX_set_key_length(
        EVP_CIPHER_CTX *x, int keylen
);
int EVP_CIPHER_CTX_ctrl(
        EVP_CIPHER_CTX *ctx, int type, int arg, void *ptr
);
int EVP_CIPHER_CTX_cleanup(
        EVP_CIPHER_CTX *a
);
const EVP_CIPHER *EVP_get_cipherbyname(
        const char *name
);
#define EVP_get_cipherbynid(a) EVP_get_cipherbyname(OBJ_nid2sn(a))
#define EVP_get_cipherbyobj(a) EVP_get_cipherbynid(OBJ_obj2nid(a))
#define EVP_CIPHER_nid(e)((e)->nid)
#define EVP_CIPHER_block_size(e)((e)->block_size)
#define EVP_CIPHER_key_length(e)((e)->key_len)
#define EVP_CIPHER_iv_length(e)((e)->iv_len)
#define EVP_CIPHER_flags(e)((e)->flags)
#define EVP_CIPHER_mode(e)((e)->flags) & EVP_CIPH_MODE)
int EVP_CIPHER_type(
        const EVP_CIPHER *ctx
);
#define EVP_CIPHER_CTX_cipher(e)((e)->cipher)
#define EVP_CIPHER_CTX_nid(e)((e)->cipher->nid)
#define EVP_CIPHER_CTX_block_size(e)((e)->cipher->block_size)
#define EVP_CIPHER_CTX_key_length(e)((e)->key_len)
#define EVP_CIPHER_CTX_iv_length(e)((e)->cipher->iv_len)
#define EVP_CIPHER_CTX_get_app_data(e)((e)->app_data)
#define EVP_CIPHER_CTX_set_app_data(e,d) ((e)->app_data=(char *)(d))
#define EVP_CIPHER_CTX_type(c) EVP_CIPHER_type(EVP_CIPHER_CTX_cipher(c))
#define EVP_CIPHER_CTX_flags(e)((e)->cipher->flags)
#define EVP_CIPHER_CTX_mode(e)((e)->cipher->flags & EVP_CIPH_MODE)
int EVP_CIPHER_param_to_asn1(
        EVP_CIPHER_CTX *c, ASN1_TYPE *type
);
int EVP_CIPHER_asn1_to_param(
        EVP_CIPHER_CTX *c, ASN1_TYPE *type
);
```

DESCRIPTION

The EVP cipher routines are a high level interface to certain symmetric ciphers.

The EVP_EncryptInit() function initializes a cipher context ctx for encryption with cipher type. The type is usually supplied by a function such as EVP_des_cbc().The key is the symmetric key to use, and iv is the IV to use (if necessary). The actual number of bytes used for the key and IV depends on the cipher. It is possible to set all parameters to NULL except type in an initial call and supply the remaining parameters in subsequent calls, all of which have type set to NULL. This is done when the default cipher parameters are not appropriate.

The EVP_EncryptUpdate() function encrypts inl bytes from the buffer in and writes the encrypted version to out. This function can be called multiple times to encrypt successive blocks of data. The amount of data written depends on the block alignment of the encrypted data. As a result, the amount of data written may be anything from zero bytes to (inl + cipher_block_size - 1); so outl should contain sufficient room. The actual number of bytes written is placed in outl.

The EVP_EncryptFinal() function encrypts the final data, that is any data that remains in a partial block. It uses standard block padding (PKCS padding). The encrypted final data is written to out which should have sufficient space for one cipher block. The number of bytes written is placed in outl. After this function is called the encryption operation is finished and no further calls to EVP_EncryptUpdate() should be made.

The EVP_DecryptInit(), EVP_DecryptUpdate(), and EVP_DecryptFinal() functions are the corresponding decryption operations. The EVP_DecryptFinal() function will return an error code if the final block is not formatted correctly. The parameters and restrictions are identical to the encryption operations except that the decrypted data buffer out passed to EVP_DecryptUpdate() should have sufficient room for (inl + cipher_block_size) bytes unless the cipher block size is 1 in which case inl bytes is sufficient.

The EVP_CipherInit(), EVP_CipherUpdate(), and EVP_CipherFinal()functions can be used for decryption or encryption. The operation performed depends on the value of the enc parameter. It should be set to 1 for encryption, 0 for decryption and -1 to leave the value unchanged (the actual value of enc being supplied in a previous call).

The EVP_CIPHER_CTX_cleanup() function clears all information from a cipher context. It should be called after all operations using a cipher are complete so sensitive information does not remain in memory.

The EVP_get_cipherbyname(), EVP_get_cipherbynid(), and EVP_get_cipherbyobj() functions return an EVP_CIPHER structure when passed a cipher name, a NID or an ASN1_OBJECT structure.

The EVP_CIPHER_nid() and EVP_CIPHER_CTX_nid() functions return the NID of a cipher when passed an EVP_CIPHER or EVP_CIPHER_CTX structure. The actual NID value is an internal value which may not have a corresponding OBJECT IDENTIFIER.

The EVP_CIPHER_key_length() and EVP_CIPHER_CTX_key_length() function return the key length of a cipher when passed an EVP_CIPHER or EVP_CIPHER_CTX structure. The constant EVP_MAX_KEY_LENGTH is the maximum key length for all ciphers. Although the EVP_CIPHER_key_length() function is fixed for a given cipher, the value of the EVP_CIPHER_CTX_key_length() function may be different for variable key length ciphers.

The EVP_CIPHER_CTX_set_key_length() function sets the key length of the cipher ctx. If the cipher is a fixed length cipher then attempting to set the key length to any value other than the fixed value is an error.

The EVP_CIPHER_iv_length() and EVP_CIPHER_CTX_iv_length() functions return the IV length of a cipher when passed an EVP_CIPHER or EVP_CIPHER_CTX. It will return zero if the cipher does not use an IV. The constant EVP_MAX_IV_LENGTH is the maximum IV length for all ciphers.

The EVP_CIPHER_block_size() and EVP_CIPHER_CTX_block_size() functions return the block size of a cipher when passed an EVP_CIPHER or EVP_CIPHER_CTX structure. The constant EVP_MAX_IV_LENGTH is also the maximum block length for all ciphers.

The EVP_CIPHER_type() and EVP_CIPHER_CTX_type() functions return the type of the passed cipher or context. This type is the actual NID of the cipher OBJECT IDENTIFIER. As such, it ignores the cipher parameters. and 40 bit RC2 and 128 bit RC2 have the same NID. If the cipher does not have an object identifier or does not have ASN1 support this function will return NID_undef.

The EVP_CIPHER_CTX_cipher() function returns the EVP_CIPHER structure when passed an EVP_CIPHER_CTX structure.

The EVP_CIPHER_mode() and EVP_CIPHER_CTX_mode() functions return the block cipher mode: EVP_CIPH_ECB_MODE, EVP_CIPH_CBC_MODE, EVP_CIPH_CFB_MODE, or EVP_CIPH_OFB_MODE. If the cipher is a stream cipher then EVP_CIPH_STREAM_CIPHER is returned.

The EVP_CIPHER_param_to_asn1() function sets the AlgorithmIdentifier parameter based on the passed cipher. This typically will include any parameters and an IV. The cipher IV (if any) must be set when this call is made. This call should be made before the cipher is actually used (before any EVP_EncryptUpdate() or EVP_DecryptUpdate() calls, for example). This function may fail if the cipher does not have any ASN1 support.

The EVP_CIPHER_asn1_to_param() function sets the cipher parameters based on an ASN1 AlgorithmIdentifier parameter. The precise effect depends on the cipher In the case of RC2, for example, it will set the IV and effective key length. This function should be called after the base cipher type is set but before the key is set. For example, the EVP_CipherInit() function will be called with the IV and key set to NULL. The EVP_CIPHER_asn1_to_param() function will be called and finally the EVP_CipherInit() function. All parameters except the key are set to NULL. It is possible for this function to fail if the cipher does not have any ASN1 support or the parameters cannot be set (for example the RC2 effective key length is not supported).

The EVP_CIPHER_CTX_ctrl() function allows various cipher specific parameters to be determined and set. Currently only the RC2 effective key length and the number of rounds of RC5 can be set.

Cipher Listing

All algorithms have a fixed key length unless otherwise stated.

EVP_enc_null()

Null cipher: does nothing.

EVP_des_cbc(void) EVP_des_ecb(void) EVP_des_cfb(void) EVP_des_ofb(void)

DES in CBC, ECB, CFB and OFB modes respectively.

EVP_des_ede_cbc(void) EVP_des_ede() EVP_des_ede_ofb(void) EVP_des_ede_cfb(void)

Two key triple DES in CBC, ECB, CFB and OFB modes respectively.

EVP_des_ede3_cbc(void) EVP_des_ede3() EVP_des_ede3_ofb(void) EVP_des_ede3_cfb(void)

Three key triple DES in CBC, ECB, CFB and OFB modes respectively.

EVP_desx_cbc(void)

DESX algorithm in CBC mode.

EVP_rc4(void)

RC4 stream cipher. This is a variable key length cipher with default key length 128 bits.

EVP_rc4_40(void)

RC4 stream cipher with 40 bit key length. This is obsolete and new code should use the EVP_rc4()and the EVP_CIPHER_CTX_set_key_length() functions.

EVP_idea_cbc() EVP_idea_ecb(void) EVP_idea_cfb(void) EVP_idea_ofb(void) EVP_idea_cbc(void)

IDEA encryption algorithm in CBC, ECB, CFB and OFB modes respectively.

EVP_rc2_cbc(void) EVP_rc2_ecb(void) EVP_rc2_cfb(void) EVP_rc2_ofb(void)

RC2 encryption algorithm in CBC, ECB, CFB and OFB modes respectively. This is a variable key length cipher with an additional parameter called effective key bits or effective key length. By default both are set to 128 bits.

EVP_rc2_40_cbc(void) EVP_rc2_64_cbc(void)

RC2 algorithm in CBC mode with a default key length and effective key length of 40 and 64 bits. These are obsolete and new code should use the EVP_rc2_cbc(), EVP_CIPHER_CTX_set_key_length(), and EVP_CIPHER_CTX_ctrl() functions to set the key length and effective key length.

EVP_bf_cbc(void) EVP_bf_ecb(void) EVP_bf_cfb(void) EVP_bf_ofb(void)

Blowfish encryption algorithm in CBC, ECB, CFB and OFB modes respectively. This is a variable key length cipher.

EVP_cast5_cbc (void) EVP_cast5_ecb(void) EVP_cast5_cfb(void) EVP_cast5_ofb(void)

CAST encryption algorithm in CBC, ECB, CFB and OFB modes respectively. This is a variable key length cipher.

EVP_rc5_32_12_16_ cbc(void) EVP_rc5_32_12_16_ ecb(void) EVP_rc5_32_12_16_ cfb(void) EVP_rc5_32_12_16_ ofb(void)

RC5 encryption algorithm in CBC, ECB, CFB and OFB modes respectively. This is a variable key length cipher with an additional "number of rounds parameter. By default the key length is set to 128 bits and 12 rounds.

NOTES

Where possible the EVP interface to symmetric ciphers should be used in preference to the low level interfaces. This is because the code then becomes transparent to the cipher used and much more flexible.

PKCS padding works by adding n padding bytes of value n to make the total length of the encrypted data a multiple of the block size. Padding is always added so if the data is already a multiple of the block size n will equal the block size. For example, if the block size is 8 and 11 bytes are to be encrypted then 5 padding bytes of value 5 will be added.

When decrypting, the final block is checked to see if it has the correct form.

Although the decryption operation can produce an error, it is not a strong test that the input data or key is correct. A random block has better than a 1-in- 256 chance of being of the correct format. Problems with the input data earlier on will not produce a final decrypt error.

The EVP_EncryptInit(), EVP_EncryptUpdate(), EVP_EncryptFinal(), EVP_DecryptInit(), EVP_DecryptUpdate(), EVP_CipherInit(), EVP_CipherUpdate(), and EVP_CIPHER_CTX_cleanup() functions did not return errors in OpenSSL version 0.9.5a or earlier. Software only versions of encryption algorithms will never return error codes for these functions, unless there is a programming error (for example, an attempt to set the key before the cipher is set in EVP_EncryptInit()).

RESTRICTIONS

For RC5 the number of rounds can be set only to 8, 12 or 16. This is a limitation of the current RC5 code rather than the EVP interface.

It is not possible to disable PKCS padding.

EVP_MAX_KEY_LENGTH and EVP_MAX_IV_LENGTH only refer to the internal ciphers with default key lengths. If custom ciphers exceed these values the results are unpredictable. This is because it has become standard practice to define a generic key as a fixed unsigned char array containing EVP_MAX_KEY_LENGTH bytes.

The ASN1 code is incomplete (and sometimes inaccurate). It has only been tested for certain common S/MIME ciphers (RC2, DES, triple DES) in CBC mode.

RETURN VALUES

The `EVP_EncryptInit()`, `EVP_EncryptUpdate()`, and `EVP_EncryptFinal()` functions return 1 for success and 0 for failure.

The `EVP_DecryptInit()` and `EVP_DecryptUpdate()` functions return 1 for success and 0 for failure. The `EVP_DecryptFinal()` function returns 0 if the decrypt failed or 1 for success.

The `EVP_CipherInit()` and `EVP_CipherUpdate()` functions return 1 for success and 0 for failure. The `EVP_CipherFinal()` function returns 1 for a decryption failure or 1 for success.

The `EVP_CIPHER_CTX_cleanup()` function returns 1 for success and 0 for failure.

The `EVP_get_cipherbyname()`, `EVP_get_cipherbynid()`, and `EVP_get_cipherbyobj()` functions return an `EVP_CIPHER` structure or NULL on error.

The `EVP_CIPHER_nid()` and `EVP_CIPHER_CTX_nid()` functions return a NID.

The `EVP_CIPHER_block_size()` and `EVP_CIPHER_CTX_block_size()` functions return the block size.

The `EVP_CIPHER_key_length()` and `EVP_CIPHER_CTX_key_length()` functions return the key length.

The `EVP_CIPHER_iv_length()` and `EVP_CIPHER_CTX_iv_length()` functions return the IV length or zero if the cipher does not use an IV.

The `EVP_CIPHER_type()` and `EVP_CIPHER_CTX_type()` functions return the NID of the cipher's OBJECT IDENTIFIER or NID_undef if it has no defined OBJECT IDENTIFIER.

The `EVP_CIPHER_CTX_cipher()` function returns an `EVP_CIPHER` structure.

The `EVP_CIPHER_param_to_asn1()` and `EVP_CIPHER_asn1_to_param()` functions return 1 for success or zero for failure.

EXAMPLES

Get the number of rounds used in RC5:

```
int nrounds;
EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GET_RC5_ROUNDS, 0, &i);
```

Get the RC2 effective key length:

```
int key_bits;
EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GET_RC2_KEY_BITS, 0, &i);
```

Set the number of rounds used in RC5:

```
int nrounds;
EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_SET_RC5_ROUNDS, i, NULL);
```

Set the number of rounds used in RC2:

```
int nrounds;
EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_SET_RC2_KEY_BITS, i, NULL);
```

SEE ALSO

Functions: *evp*

EVP_OpenInit

NAME

EVP_OpenInit, EVP_OpenUpdate, EVP_OpenFinal – EVP envelope decryption

SYNOPSIS

```
#include <openssl/evp.h>

int EVP_OpenInit(

        EVP_CIPHER_CTX *ctx,EVP_CIPHER *type,unsigned char *ek, int ekl,unsigned char
        *iv,EVP_PKEY *priv

);

int EVP_OpenUpdate(

        EVP_CIPHER_CTX *ctx, unsigned char *out, int *outl, unsigned char *in, int inl

);

int EVP_OpenFinal(

        EVP_CIPHER_CTX *ctx, unsigned char *out, int *outl

);
```

DESCRIPTION

The EVP envelope routines are a high level interface to envelope decryption. They decrypt a public key encrypted symmetric key and then decrypt data using it.

The `EVP_OpenInit()` function initializes a cipher context `ctx` for decryption with cipher `type`. It decrypts the encrypted symmetric key of length `ekl` bytes passed in the `ek` parameter using the private key, `priv`. The IV is supplied in the `iv` parameter.

The `EVP_OpenUpdate()` and `EVP_OpenFinal()` functions have the same properties as the `EVP_DecryptUpdate()` and `EVP_DecryptFinal()` functions, as documented on the ***EVP_EncryptInit*** reference page.

NOTES

It is possible to call `EVP_OpenInit()` twice in the same way as `EVP_DecryptInit()`. The first call should have `priv` set to NULL and (after setting any cipher parameters) it should be called again with `type` set to NULL.

If the cipher passed in the `type` parameter is a variable length cipher then the key length will be set to the value of the recovered key length. If the cipher is a fixed length cipher then the recovered key length must match the fixed cipher length.

RETURN VALUES

The `EVP_OpenInit()` function returns 0 on error or a nonzero integer (actually the recovered secret key size) if successful.

The `EVP_OpenUpdate()` function returns 1 for success or 0 for failure.

The `EVP_OpenFinal()` function returns 0 if the decrypt failed or 1 for success.

EVP_SealInit

NAME

EVP_SealInit, EVP_SealUpdate, EVP_SealFinal – EVP envelope encryption

SYNOPSIS

```
#include <openssl/evp.h>

int EVP_SealInit(

        EVP_CIPHER_CTX *ctx, EVP_CIPHER *type, unsigned char **ek, int *ekl, unsigned
        char *iv,EVP_PKEY **pubk, int npubk

);

int EVP_SealUpdate(

        EVP_CIPHER_CTX *ctx, unsigned char *out, int *outl, unsigned char *in, int inl

);

int EVP_SealFinal(

        EVP_CIPHER_CTX *ctx, unsigned char *out, int *outl

);
```

DESCRIPTION

The EVP envelope routines are a high level interface to envelope encryption. They generate a random key and then envelope it by using public key encryption. Data can then be encrypted using this key.

The `EVP_SealInit()` function initializes a cipher context `ctx` for encryption with cipher `type` using a random secret key and IV supplied in the `iv` parameter. The `type` is normally supplied by a function such as `EVP_des_cbc()`. The secret key is encrypted using one or more public keys. This allows the same encrypted data to be decrypted using any of the corresponding private keys. The `ek` is an array of buffers where the public key encrypted secret key will be written. Each buffer must contain enough room for the corresponding encrypted key: that is, `ek[i]` must have room for `EVP_PKEY_size(pubk[i])` bytes. The actual size of each encrypted secret key is written to the array `ekl`. The `pubk` is an array of `npubk` public keys.

The `EVP_SealUpdate()` and `EVP_SealFinal()` functions have the same properties as the `EVP_EncryptUpdate()` and `EVP_EncryptFinal()` functions, as documented on the ***EVP_EncryptInit*** reference page.

NOTES

Because a random secret key is generated the random number generator must be seeded before calling `EVP_SealInit()`.

The public key must be RSA because it is the only OpenSSL public key algorithm that supports key transport.

Envelope encryption is the usual method of using public key encryption on large amounts of data. This is because public key encryption is slow but symmetric encryption is fast. So symmetric encryption is used for bulk encryption and the small random symmetric key used is transferred using public key encryption.

It is possible to call `EVP_SealInit()` twice in the same way as `EVP_EncryptInit()`. The first call should have `npubk` set to 0 and (after setting any cipher parameters) it should be called again with `type` set to NULL.

RETURN VALUES

The `EVP_SealInit()` function returns 0 on error or `npubk` if successful.

The `EVP_SealUpdate()` and `EVP_SealFinal()` functions return 1 for success and 0 for failure.

SEE ALSO

Functions: *evp*, *rand*, *EVP_EncryptInit*, *EVP_OpenInit*

EVP_SignInit

NAME

EVP_SignInit, EVP_SignUpdate, EVP_SignFinal – EVP signing functions

SYNOPSIS

```
#include <openssl/evp.h>
void EVP_SignInit(
        EVP_MD_CTX *ctx, const EVP_MD *type
);
void EVP_SignUpdate(
        EVP_MD_CTX *ctx, const void *d, unsigned int cnt
);
int EVP_SignFinal(
        EVP_MD_CTX *ctx,unsigned char *sig,unsigned int *s, EVP_PKEY *pkey
);
int EVP_PKEY_size(
        EVP_PKEY *pkey
);
```

DESCRIPTION

The EVP signature routines are a high level interface to digital signatures.

The `EVP_SignInit()` function initializes a signing context `ctx` to using digest `type`. This typically will be supplied by a function such as `EVP_sha1()`.

The `EVP_SignUpdate()` function hashes `cnt` bytes of data at `d` into the signature context `ctx`. This function can be called several times on the same `ctx` to include additional data.

The `EVP_SignFinal()` function signs the data in `ctx` using the private key `pkey` and places the signature in `sig`. If the `s` parameter is not NULL then the number of bytes of data written (i.e. the length of the signature) will be written to the integer at `s`, at most EVP_PKEY_size(pkey) bytes will be written. After calling `EVP_SignFinal()`, no additional calls to `EVP_SignUpdate()` can be made, but the `EVP_SignInit()` function can be called to initialize a new signature operation.

The `EVP_PKEY_size()` function returns the maximum size of a signature in bytes. The actual signature returned by the `EVP_SignFinal()` function may be smaller.

NOTES

The `EVP` interface to digital signatures should be used in preference to the low level interfaces. This is because the code then becomes transparent to the algorithm used and much more flexible.

Due to the link between message digests and public key algorithms the correct digest algorithm must be used with the correct public key type. A list of algorithms and associated public key algorithms appears in *EVP_DigestInit*.

When signing with DSA private keys the random number generator must be seeded or the operation will fail. The random number generator does not need to be seeded for RSA signatures.

RESTRICTIONS

Several of the functions do not return values. Although the internal digest operations will never fail some future hardware based operations might.

RETURN VALUES

The `EVP_SignInit()` and `EVP_SignUpdate()` functions do not return values.

The `EVP_SignFinal()` function returns 1 for success and 0 for failure.

The `EVP_PKEY_size()` function returns the maximum size of a signature in bytes.

The error codes can be obtained by using `ERR_get_error()`.

HISTORY

The `EVP_SignInit()`, `EVP_SignUpdate()`, and `EVP_SignFinal()` functions are available in all versions of SSLeay and OpenSSL.

SEE ALSO

Commands: *digest*

Functions: *EVP_VerifyInit*, *EVP_DigestInit*, *err*, *evp*, *hmac*, *md2*, *md5*, *mdc2*, *ripemd160*, *sha*

EVP_VerifyInit

NAME

EVP_VerifyInit, EVP_VerifyUpdate, EVP_VerifyFinal – EVP signature verification functions

SYNOPSIS

```
#include <openssl/evp.h>
void EVP_VerifyInit(
        EVP_MD_CTX *ctx, const EVP_MD *type
);
void EVP_VerifyUpdate(
        EVP_MD_CTX *ctx, const void *d, unsigned int cnt
);
int EVP_VerifyFinal(
        EVP_MD_CTX *ctx,unsigned char *sigbuf, unsigned int siglen,EVP_PKEY *pkey
);
```

DESCRIPTION

The EVP signature verification routines are a high level interface to digital signatures.

The `EVP_VerifyInit()` function initializes a verification context `ctx` to using digest `type`. This will typically be supplied by a function such as `EVP_sha1()`.

The `EVP_VerifyUpdate()` function hashes `cnt` bytes of data at `d` into the verification context `ctx`. This function can be called several times on the same `ctx` to include additional data.

The `EVP_VerifyFinal()` function verifies the data in `ctx` using the public key `pkey` and against the `siglen` bytes at `sigbuf`. After calling the `EVP_VerifyFinal()` function no additional calls to the `EVP_VerifyUpdate()` function can be made, but the `EVP_VerifyInit()` function can be called to initialize a new verification operation.

NOTES

The `EVP` interface to digital signatures should be used in preference to the low level interfaces. This is because the code then becomes transparent to the algorithm used and much more flexible.

Due to the link between message digests and public key algorithms the correct digest algorithm must be used with the correct public key type. A list of algorithms and associated public key algorithms appears in *EVP_DigestInit*.

RESTRICTIONS

Several of the functions do not return values. Although the internal digest operations will never fail some future hardware based operations might.

RETURN VALUES

The `EVP_VerifyInit()` and `EVP_VerifyUpdate()` functions do not return values.

The `EVP_VerifyFinal()` function returns 1 for a correct signature, 0 for failure and -1 if some other error occurred.

The error codes can be obtained by using `ERR_get_error()`.

HISTORY

The `EVP_VerifyInit()`, `EVP_VerifyUpdate()`, and `EVP_VerifyFinal()` functions are available in all versions of SSLeay and OpenSSL.

SEE ALSO

Commands: *digest*

Functions: *evp*, *EVP_SignInit*, *EVP_DigestInit*, *err*, *evp*, *hmac*, *md2*, *md5*, *mdc2*, *ripemd160*, *sha*

gendsa

NAME

gendsa – Generate a DSA private key from a set of parameters

SYNOPSIS

`openssl gendsa [-out` *`filename`*`] [-des] [-des3] [-idea] [-rand` *`filename`*`] [-`*`paramfile`*`]`

OPTIONS

`des|des3|idea`

Encrypts the private key with the DES, triple DES, or the IDEA ciphers respectively before outputting it. A pass phrase is prompted for. If none of these options is specified no encryption is used.

`rand` *`filename`*

A file or files containing random data used to seed the random number generator, or an EGD socket. (See *RAND_egd*.) Multiple files can be specified separated by an OS-dependent character. The separator is a semicolon (;) for MS-Windows, a comma (,) for OpenVMS, and a colon (:) for all others.

`paramfile`

Specifies the DSA parameter file to use. The parameters in this file determine the size of the private key. DSA parameters can be generated and examined using the `openssl dsaparam` command.

DESCRIPTION

The `gendsa` command generates a DSA private key from a DSA parameter file (which will be typically generated by the `openssl dsaparam` command).

NOTES

DSA key generation is little more than random number generation, so it is much quicker than RSA key generation.

SEE ALSO

Commands: *dsaparam*, *dsa*, *genrsa*, *rsa*

genrsa

NAME

genrsa – Generate an RSA private key

SYNOPSIS

openssl genrsa [-out *filename*] [-passout *arg*] [-des] [-des3] [-idea] [-f4] [-3]
[-rand *filename*] [-*numbits*]

OPTIONS

out *filename*

Outputs the filename. If this argument is not specified then standard output is used.

passout *arg*

Outputs the file password source. For more information about the format of arg see the Pass Phrase Arguments section in *openssl*.

des|des3|idea

Encrypts the private key with the DES, triple DES, or the IDEA ciphers, respectively, before outputting it. If none of these options is specified no encryption is used. If encryption is used, there is a prompt for a pass phrase if it is not supplied via the passout argument.

F4|3

The public exponent to use, either 65537 or 3. The default is 65537.

rand *filename*

A file or files containing random data used to seed the random number generator, or an EGD socket. (See *RAND_egd*.) Multiple files can be specified separated by an OS-dependent character. The separator is a semicolon (;) for MS-Windows, a comma (,) for OpenVMS, and a colon (:) for all others.

numbits

The size of the private key to generate in bits. This must be the last option specified. The default is 512.

DESCRIPTION

The genrsa command generates an RSA private key.

NOTES

RSA private key generation essentially involves the generation of two prime numbers. When generating a private key, various symbols will be output to indicate the progress of the generation. A period (.) represents each number that passed an initial sieve test. A plus sign (+) means a number has passed a single round of the Miller-Rabin primality test. A newline means that the number has passed all the prime tests; the actual number depends on the key size.

Because key generation is a random process, the time taken to generate a key may vary.

RESTRICTIONS

A quirk of the prime generation algorithm is that it cannot generate small primes. Therefore, the number of bits should not be less that 64. For typical private keys this will not matter because, for security reasons, they will be much larger (typically 1024 bits).

SEE ALSO

Commands: *gendsa*

HMAC

NAME

HMAC, HMAC_Init, HMAC_Update, HMAC_Final, HMAC_cleanup – HMAC message authentication code

SYNOPSIS

```
#include <openssl/hmac.h>

unsigned char *HMAC(
        const EVP_MD *evp_md, const void *key, int key_len, const unsigned char *d, int
        n, unsigned char *md, unsigned int *md_len
);

void HMAC_Init(
        HMAC_CTX *ctx, const void *key, int key_len, const EVP_MD *md
);

void HMAC_Update(
        HMAC_CTX *ctx, const unsigned char *data, int len
);

void HMAC_Final(
        HMAC_CTX *ctx, unsigned char *md, unsigned int *len
);

void HMAC_cleanup(
        HMAC_CTX *ctx
);
```

DESCRIPTION

HMAC is a message authentication code (MAC), i.e. a keyed hash function used for message authentication, which is based on a hash function.

The `HMAC()` function computes the message authentication code of the `n` bytes at `d` using the hash function `evp_md` and the key `key` which is `key_len` bytes long.

It places the result in `md` (which must have space for the output of the hash function, which is no more than `EVP_MAX_MD_SIZE` bytes). If `md` is NULL, the digest is placed in a static array. The size of the output is placed in `md_len`, unless it is `NULL`.

The `evp_md` can be `EVP_sha1()`, `EVP_ripemd160()`, etc. The `key` and `evp_md` can be `NULL` if a key and hash function have been set in a previous call to `HMAC_Init()` for that `HMAC_CTX`.

The `HMAC_cleanup()` function erases the key and other data from the `HMAC_CTX`.

The following functions may be used if the message is not completely stored in memory:

The `HMAC_Init()` function initializes a `HMAC_CTX` structure to use the hash function `evp_md` and the key `key` which is `key_len` bytes long.

The `HMAC_Update()` function can be called repeatedly with chunks of the message to be authenticated (`len` bytes at `data`).

The `HMAC_Final()` function places the message authentication code in `md`, which must have space for the hash function output.

HMAC conforms to RFC 2104.

RETURN VALUES

The `HMAC()` function returns a pointer to the message authentication code.

The `HMAC_Init()`, `HMAC_Update()`, `HMAC_Final()`, and `HMAC_cleanup()` do not return values.

HISTORY

The `HMAC()`, `HMAC_Init()`, `HMAC_Update()`, `HMAC_Final()`, and `HMAC_cleanup()` functions are available since SSLeay 0.9.0.

SEE ALSO

Functions: *sha*, *evp*

lh_stats

NAME

lh_stats, lh_node_stats, lh_node_usage_stats, lh_stats_bio, lh_node_stats_bio, lh_node_usage_stats_bio – LHASH statistics

SYNOPSIS

```
#include <openssl/lhash.h>
void lh_stats(
       LHASH *table, FILE *out
);
void lh_node_stats(
       LHASH *table, FILE *out
);
void lh_node_usage_stats(
        LHASH *table, FILE *out
);
void lh_stats_bio(
       LHASH *table, BIO *out
);
void lh_node_stats_bio(
       LHASH *table, BIO *out
);
void lh_node_usage_stats_bio(
        LHASH *table, BIO *out
);
```

DESCRIPTION

The `LHASH` structure records statistics about most aspects of accessing the hash table. It is a legacy of Eric Young who wrote the library for the purpose of implementing a useful algorithm rather than for a particular software product.

The `lh_stats()` function prints out statistics on the size of the hash table, how many entries are in it, and the number and result of calls to the routines in this library.

The `lh_node_stats()` function prints the number of entries for each bucket in the hash table.

The `lh_node_usage_stats()` function prints out a short summary of the state of the hash table. It prints the load and the actual load. The load is the average number of data items per bucket in the hash table. The actual load is the average number of items per bucket, but only for buckets which contain entries. So the actual load is the average number of searches that will need to find an item in the hash table, while the load is the average number that will be done to record a miss.

The `lh_stats_bio()`, `lh_node_stats_bio()`, and `lh_node_usage_stats_bio()` functions are the same, except that the output goes to a `BIO`.

RETURN VALUES

These functions do not return values.

HISTORY

These functions are available in all versions of SSLeay and OpenSSL. This reference page is derived from the SSLeay documentation.

SEE ALSO

Functions: *bio*, *lhash*

lhash

NAME

lhash, lh_new, lh_free, lh_insert, lh_delete, lh_retrieve, lh_doall, lh_doall_arg, lh_error – Dynamic hash table

SYNOPSIS

```
#include <openssl/lhash.h>

LHASH *lh_new(
        unsigned long (*hash)(/*void *a*/), int (*compare)(/*void *a,void *b*/)
);

void lh_free(
        LHASH *table
);

void *lh_insert(
        LHASH *table, void *data
);

void *lh_delete(
        LHASH *table, void *data
);

void *lh_retrieve(
        LHASH *table, void *data
);

void lh_doall(
        LHASH *table, void (*func)(/*void *b*/)
);

void lh_doall_arg(
        LHASH *table, void (*func)(/*void *a,void *b*/), void *arg
);

int lh_error(
        LHASH *table
);
```

DESCRIPTION

This library implements dynamic hash tables. The hash table entries can be arbitrary structures. Usually they consist of key and value fields.

The `lh_new()` function creates a new `LHASH` structure. The `hash` takes a pointer to the structure and returns an unsigned long hash value of its key field. The hash value is normally truncated to a power of 2, so make sure that your hash function returns well mixed low order bits. The `compare` takes two arguments, and returns 0 if their keys are equal, non-zero otherwise.

The lh_free() function frees the LHASH structure table. Allocated hash table entries will not be freed; consider using the lh_doall() function to deallocate any remaining entries in the hash table.

The lh_insert() function inserts the structure pointed to by data into table. If there already is an entry with the same key, the old value is replaced. The lh_insert() function stores pointers; the data are not copied.

The lh_delete() function deletes an entry from table.

The lh_retrieve() function looks up an entry in table. Normally, data is a structure with the key field set; the function will return a pointer to a fully populated structure.

The lh_doall() function will, for every entry in the hash table, call func with the data item as parameters. This function can be quite useful when used as follows: void cleanup(STUFF *a) { STUFF_free(a); } lh_doall(hash,cleanup); lh_free(hash). This can be used to free all the entries. The lh_free() function then cleans up the 'buckets that point to nothing. When doing this, be careful if you delete entries from the hash table in func. The table might decrease in size, moving items lower in the hash table. This could cause some entries to be skipped. The best solution to this problem is to set hash->down_load=0 before you start. This will stop the hash table from decreasing in size.

The lh_doall_arg() function is the same as lh_doall() except that func will be called with arg as the second argument.

The lh_error() macro can be used to determine if an error occurred in the last operation.

Internals

The following description is based on the SSLeay documentation:

The lhash library implements a hash table described in the *Communications of the ACM* in 1991. What makes this hash table different is that as the table fills, the hash table is increased (or decreased) in size via the OPENSSL_realloc() function. When a resize is done, instead of all hashes being redistributed over twice as many buckets, one bucket is split. So when an expand is done, there is only a minimal cost to redistribute some values. Subsequent inserts will cause more single bucket redistributions but there will never be a sudden large cost due to redistributing all the buckets.

The state for a particular hash table is kept in the LHASH structure. The decision to increase or decrease the hash table size is made depending on the load of the hash table. The load is the number of items in the hash table divided by the size of the hash table. The default values are as follows:

- if (hash->up_load < load) => expand
- if (hash->down_load > load) => contract

The up_load has a default value of 1, and down_load has a default value of 2. These numbers can be modified by the application by adjusting the up_load and down_load variables. The load is kept in a form which is multiplied by 256. So hash->up_load=8*256; will cause a load of 8 to be set.

If you are interested in performance, the field to watch is num_comp_calls. The hash library keeps track of the hash value for each item so when a lookup is done, the hashes are compared. If there is a match, then a full compare is done, and hash->num_comp_calls is incremented. If num_comp_calls is not equal to num_delete plus num_retrieve it means that your hash function is generating hashes that are the same for different values. It is probably worth changing your hash function if this is the case because even if your hash table has 10 items in a bucket, it can be searched with 10 unsigned long compares and 10 linked list traverses. This will be much less expensive that 10 calls to your compare function.

The lh_strhash() is a demo string hashing function:

```
unsigned long lh_strhash(const char *c);
```

Since the `LHASH` routines would normally be passed structures, this routine would not normally be passed to `lh_new()`, rather it would be used in the function passed to the `lh_new()` function.

RESTRICTIONS

The `lh_insert()` function returns `NULL` both for success and error.

RETURN VALUES

The `lh_new()` function returns `NULL` on error, otherwise a pointer to the new `LHASH` structure.

When a hash table entry is replaced, the `lh_insert()` function returns the value being replaced. `NULL` is returned on normal operation and on error.

The `lh_delete()` function returns the entry being deleted. `NULL` is returned if there is no such value in the hash table.

The `lh_retrieve()` function returns the hash table entry if it has been found, `NULL` otherwise.

The `lh_error()` function returns 1 if an error occurred in the last operation, 0 otherwise.

The `lh_free()`, `lh_doall()`, and `lh_doall_arg()` functions return no values.

HISTORY

The `lhash` library is available in all versions of SSLeay and OpenSSL. The `lh_error()` function was added in SSLeay 0.9.1b. This reference page is derived from the SSLeay documentation.

SEE ALSO

Functions: *lh_stats*

md5

NAME

md5: MD2, MD4, MD5, MD2_Init, MD2_Update, MD2_Final, MD4_InitMD4_Update, MD4_Final, MD5_Init, MD5_Update, MD5_Final – MD2, MD4, and MD5 hash functions

SYNOPSIS

```
#include <openssl/md2.h>
unsigned char *MD2(
       const unsigned char *d, unsigned long n, unsigned char *md
);
void MD2_Init(
       MD2_CTX *c
);
void MD2_Update(
       MD2_CTX *c, const unsigned char *data, unsigned long len
);
void MD2_Final(
       unsigned char *md, MD2_CTX *c
);
#include <openssl/md4.h>
unsigned char *MD4(
       const unsigned char *d, unsigned long n, unsigned char *md
);
void MD4_Init(
       MD4_CTX *c
);
void MD4_Update(
       MD4_CTX *c, const void *data, unsigned long len
);
void MD4_Final(
       unsigned char *md, MD4_CTX *c
);
#include <openssl/md5.h>
unsigned char *MD5(
       const unsigned char *d, unsigned long n, unsigned char *md
);
void MD5_Init(
       MD5_CTX *c
);
```

```
void MD5_Update(
        MD5_CTX *c, const void *data, unsigned long len
);
void MD5_Final(
        unsigned char *md, MD5_CTX *c
);
```

DESCRIPTION

MD2, MD4, and MD5 are cryptographic hash functions with a 128 bit output.

The `MD2()`, `MD4()`, and `MD5()` functions compute the MD2, MD4, and MD5 message digest of the `n` bytes at `d` and place it in `md` (which must have space for MD2_DIGEST_LENGTH == MD4_DIGEST_LENGTH == MD5_DIGEST_LENGTH == 16 bytes of output). If `md` is NULL, the digest is placed in a static array.

The following functions may be used if the message is not completely stored in memory:

The `MD2_Init()` function initializes a `MD2_CTX` structure.

The `MD2_Update()` function can be called repeatedly with chunks of the message to be hashed (`len` bytes at `data`).

The `MD2_Final()` function places the message digest in `md`, which must have space for MD2_DIGEST_LENGTH == 16 bytes of output, and erases the `MD2_CTX`.

The `MD4_Init()`, `MD4_Update()`, `MD4_Final()`, `MD5_Init()`, `MD5_Update()`, and `MD5_Final()` functions are analogous using an `MD4_CTX` and `MD5_CTX` structure.

Applications should use the higher level functions, such as `EVP_DigestInit()`, instead of calling the hash functions directly.

MD2, MD4, and MD5 conform to RFC 1319, RFC 1320, and RFC 1321.

NOTES

MD2, MD4, and MD5 are recommended only for compatibility with existing applications. In new applications, SHA-1 or RIPEMD-160 are preferred.

RETURN VALUES

The `MD2()`, `MD4()`, and `MD5()` functions return pointers to the hash value.

The `MD2_Init()`, `MD2_Update()`, `MD2_Final()`, `MD4_Init()`, `MD4_Update()`, `MD4_Final()`, `MD5_Init()`, `MD5_Update()`, and `MD5_Final()` functions do not return values.

HISTORY

The `MD2()`, `MD2_Init()`, `MD2_Update()`, `MD2_Final()`), `MD5()`, `MD5_Init()`, `MD5_Update()`, and `MD5_Final()` functions are available in all versions of SSLeay and OpenSSL.

The `MD4()`, `MD4_Init()`, and `MD4_Update()` functions are available in OpenSSL 0.9.6 and above.

SEE ALSO

Functions: *sha*, *ripemd160*, *EVP_DigestInit*

MDC2

NAME

MDC2, MDC2_Init, MDC2_Update, MDC2_Final – MDC2 hash function

SYNOPSIS

```
#include <openssl/mdc2.h>
unsigned char *MDC2(
        const unsigned char *d, unsigned long n, unsigned char *md
);
void MDC2_Init(
        MDC2_CTX *c
);
void MDC2_Update(
        MDC2_CTX *c, const unsigned char *data, unsigned long len
);
void MDC2_Final(
        unsigned char *md, MDC2_CTX *c
);
```

DESCRIPTION

MDC2 is a method to construct hash functions with 128 bit output from block ciphers. These functions are an implementation of MDC2 with DES.

The `MDC2()` function computes the MDC2 message digest of the `n` bytes at `d` and places it in `md` (which must have space for MDC2_DIGEST_LENGTH == 16 bytes of output). If `md` is NULL, the digest is placed in a static array.

The following functions can be used if the message is not completely stored in memory:

The `MDC2_Init()` function initializes a `MDC2_CTX` structure.

The `MDC2_Update()` function can be called repeatedly with chunks of the message to be hashed (`len` bytes at `data`).

The `MDC2_Final()` function places the message digest in `md`, which must have space for MDC2_DIGEST_LENGTH == 16 bytes of output, and erases the `MDC2_CTX`.

Applications should use the higher level functions, such as `EVP_DigestInit()`, instead of calling the hash functions directly.

MDC2 conforms to ISO/IEC 10118-2, with DES.

RETURN VALUES

The `MDC2()` function returns a pointer to the hash value.

The `MDC2_Init()`, `MDC2_Update()`, and `MDC2_Final()` functions do not return values.

HISTORY

The `MDC2()`, `MDC2_Init()`, `MDC2_Update()`, and `MDC2_Final()` functions are available since SSLeay 0.8.

SEE ALSO

Functions: *sha*, *EVP_DigestInit*

nseq

NAME

nseq – Create or examine a netscape certificate sequence

SYNOPSIS

`openssl nseq [-in filename] [-out filename] [-toseq]`

OPTIONS

`in` *filename*

Specifies the input filename to read or standard input if this option is not specified.

`out` *filename*

Specifies the output filename or standard output by default.

`toseq`

Normally a Netscape certificate sequence will be input and the output is the certificates contained in it. With the `toseq` option the situation is reversed; a Netscape certificate sequence is created from a file of certificates.

DESCRIPTION

The `nseq` command takes a file containing a Netscape certificate sequence and prints out the certificates contained in it or takes a file of certificates and converts it into a Netscape certificate sequence.

NOTES

The `PEM` encoded form uses the same headers and footers as a certificate:

```
-----BEGIN CERTIFICATE-----
-----END CERTIFICATE-----
```

A Netscape certificate sequence is a Netscape specific form that can be sent to browsers as an alternative to the standard PKCS#7 format when several certificates are sent to the browser: for example during certificate enrollment. It is used by Netscape certificate server for example.

RESTRICTIONS

This program needs a few more options, such as allowing DER or PEM input and output files and allowing multiple certificate files to be used.

EXAMPLES

Output the certificates in a Netscape certificate sequence

```
openssl nseq -in nseq.pem -out certs.pem
```

Create a Netscape certificate sequence

```
openssl nseq -in certs.pem -toseq -out nseq.pem
```

openssl

NAME

openssl – OpenSSL command line tool

SYNOPSIS

`openssl` *command* `[-`*command_opts*`] [-`*command_args*`]`

`openssl [-list-standard-commands | list-message-digest-commands | list-cipher-commands]`

`openssl no-XXX [-`*arbitrary options*`]`

COMMAND SUMMARY

The `openssl` program provides a rich variety of commands, each of which often has a wealth of options and arguments.

The pseudo-commands `list-standard-commands`, `list-message-digest-commands`, and `list-cipher-commands` output a list (one entry per line) of the names of all standard commands, message digest commands, or cipher commands, respectively, that are available in the present `openssl` utility.

The pseudo-command `no-XXX` tests whether a command of the specified name is available. If no command named *XXX* exists, it returns 0 (success) and prints `no-XXX`; otherwise it returns 1 and prints `XXX`. In both cases, the output goes to `stdout` and nothing is printed to `stderr`. Additional command line arguments are always ignored. Since for each cipher there is a command of the same name, this provides an easy way for shell scripts to test for the availability of ciphers in the `openssl` program. (The `no-XXX` command is not able to detect pseudo-commands such as `quit`, `list- ...commands`, or `no-XXX` itself.)

STANDARD COMMANDS

`asn1parse`

Parse an ASN.1 sequence.

`ca`

Certificate Authority (CA) Management.

`ciphers`

Cipher Suite Description Determination.

`crl`

Certificate Revocation List (CRL) Management.

`crl2pkcs7`

CRL to PKCS#7 Conversion.

`dgst`

Message Digest Calculation.

`dh`

Diffie-Hellman Parameter Management. Obsoleted by `dHParam`.

`dsa`

DSA Data Management.

`dsaparam`

DSA Parameter Generation.

`enc`

Encoding with Ciphers.

`errstr`

Error Number to Error String Conversion.

`dHParam`

Generation and Management of Diffie-Hellman Parameters.

`gendh`

Generation of Diffie-Hellman Parameters. Obsoleted by `dHParam`.

`gendsa`

Generation of DSA Parameters.

`genrsa`

Generation of RSA Parameters.

`passwd`

Generation of hashed passwords.

`pkcs12`

PKCS#12 Data Management.

`pkcs7`

PKCS#7 Data Management.

`rand`

Generate pseudo-random bytes.

`req`

X.509 Certificate Signing Request (CSR) Management.

`rsa`

RSA Data Management.

`rsautl`

RSA utility for signing, verification, encryption, and decryption.

`s_client`

Implements a generic SSL/TLS client which can establish a transparent connection to a remote server speaking SSL/TLS. It is intended for testing purposes only and provides only rudimentary interface functionality but internally uses mostly all functionality of the OpenSSL `ssl` library.

`s_server`

This implements a generic SSL/TLS server which accepts connections from remote clients speaking SSL/TLS. It is intended for testing purposes only and provides only rudimentary interface functionality but internally uses mostly all functionality of the OpenSSL `ssl` library. It provides both an own command line oriented protocol for testing SSL functions and a simple HTTP response facility to emulate an SSL/TLS-aware webserver.

`s_time`

SSL Connection Timer.

`sess_id`

SSL Session Data Management.

`smime`

S/MIME mail processing.

`speed`

Algorithm Speed Measurement.

`verify`

X.509 Certificate Verification.

`version`

OpenSSL Version Information.

`x509`

X.509 Certificate Data Management.

MESSAGE DIGEST COMMANDS

`md2`

MD2 Digest

`md5`

MD5 Digest

`mdc2`

MDC2 Digest

`rmd160`

RMD-160 Digest

`sha`

SHA Digest

`sha1`

SHA-1 Digest

ENCODING AND CIPHER COMMANDS

`base64`

Base64 Encoding

`bf bf-cbc bf-cfb bf-ecb bf-ofb`

Blowfish Cipher

`cast cast-cbc`

CAST Cipher

`cast5-cbc cast5-cfb cast5-ecb cast5-ofb`

CAST5 Cipher

`des des-cbc des-cfb des-ecb des-ede des-ede-cbc des-ede-cfb des-ede-ofb des-ofb`

DES Cipher

`des3 desx des-ede3 des-ede3-cbc des-ede3-cfb des-ede3-ofb`

Triple-DES Cipher

`idea idea-cbc idea-cfb idea-ecb idea-ofb`

IDEA Cipher

`rc2 rc2-cbc rc2-cfb rc2-ecb rc2-ofb`

RC2 Cipher

`rc4`

RC4 Cipher

`rc5 rc5-cbc rc5-cfb rc5-ecb rc5-ofb`

RC5 Cipher

PASSWORD PHRASE ARGUMENTS

Several commands accept password arguments, typically using the `passin` and the `passout` options for input and output passwords respectively. These allow the password to be obtained from a variety of sources. Both of these options take a single argument whose format is described below. If no password argument is given and a password is required then the user is prompted to enter one. This will typically be read from the current terminal with echoing turned off.

`pass:password`

The actual password is `password`. Since the password is visible to utilities (such as `ps` under UNIX), this form should only be used where security is not important.

`env:var`

Obtains the password from the environment variable `var`. Since the environment of other processes is visible on certain platforms (e.g. `ps` under certain UNIX operating systems), this option should be used with caution.

`file:pathname`

The first line of `pathname` is the password. If the same `pathname` argument is supplied to `passin` and `passout` arguments then the first line will be used for the input password and the next line for the output password. The `pathname` need not refer to a regular file. It could, for example, refer to a device or named pipe.

`fd:number`

Reads the password from the file descriptor `number`. This can be used, for example, to send the data via a pipe.

`stdin`

Reads the password from standard input.

DESCRIPTION

OpenSSL is a cryptography toolkit implementing the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) network protocols and related cryptography standards required by them.

The `openssl` program is a command line tool for using the various cryptography functions of OpenSSL's `crypto` library from the shell. It can be used for the following:

- Creation of RSA, DH and DSA key parameters
- Creation of X.509 certificates, CSRs and CRLs
- Calculation of Message Digests
- Encryption and Decryption with Ciphers
- SSL/TLS Client and Server Tests
- Handling of S/MIME signed or encrypted mail

HISTORY

The *openssl* document appeared in OpenSSL 0.9.2. The `list-XXXcommands` pseudo-commands were added in OpenSSL 0.9.3. The `no-XXX` pseudo-commands were added in OpenSSL 0.9.5a. For notes on the availability of other commands, see their individual manual pages.

SEE ALSO

Commands: *asn1parse*, *ca*, *crl*, *crl2pkcs7*, *dgst*, *dHParam*, *dsa*, *dsaparam*, *enc*, *gendsa*, *genrsa*, *nseq*, *openssl*, *passwd*, *pkcs12*, *pkcs7*, *pkcs8*, *rand*, *req*, *rsa*, *rsautl*, *s_client*, *s_server*, *smime*, *spkac*, *verify*, *version*, *x509*

Functions: *crypto*, *ssl*

Others: *config*

OpenSSL_add_all_algorithms

NAME

OpenSSL_add_all_algorithms, OpenSSL_add_all_ciphers, OpenSSL_add_all_digests – Add algorithms to internal table

SYNOPSIS

```
#include <openssl/evp.h>
void OpenSSL_add_all_algorithms(
        void
);
void OpenSSL_add_all_ciphers(
        void
);
void OpenSSL_add_all_digests(
        void
);
void EVP_cleanup(
       void
);
```

DESCRIPTION

OpenSSL keeps an internal table of digest algorithms and ciphers. It uses this table to lookup ciphers via functions such as `EVP_get_cipher_byname()`.

The `OpenSSL_add_all_digests()` function adds all digest algorithms to the table.

The `OpenSSL_add_all_algorithms()` function adds all algorithms to the table (digests and ciphers).

The `OpenSSL_add_all_ciphers()` function adds all encryption algorithms to the table including password based encryption algorithms.

The `EVP_cleanup()` function removes all ciphers and digests from the table.

NOTES

A typical application will call the `OpenSSL_add_all_algorithms()` function initially and the `EVP_cleanup()` function before exiting.

An application does not need to add algorithms to use them explicitly, for example by `EVP_sha1()`. It needs to add them if it (or any of the functions it calls) needs to lookup algorithms.

The cipher and digest lookup functions are used in many parts of the library. If the table is not initialized several functions will not work correctly and complain they cannot find algorithms. This includes the PEM, PKCS#12, SSL and S/MIME libraries. This is a common query in the OpenSSL mailing lists.

Calling the `OpenSSL_add_all_algorithms()` function links all algorithms. As a result, a statically linked executable can be quite large. If this is important, it is possible to add only the required ciphers and digests.

RESTRICTIONS

Although the functions do not return error codes it is possible for them to fail. This will only happen as a result of a memory allocation failure, so it is not much of a problem in practice.

RETURN VALUES

None of the functions return a value.

SEE ALSO

Functions: *evp*, *EVP_DigestInit*, *EVP_EncryptInit*

OPENSSL_VERSION_NUMBER

NAME

OPENSSL_VERSION_NUMBER, SSLeay, SSLeay_version – Get OpenSSL version number

SYNOPSIS

```
#include <openssl/opensslv.h>
#define OPENSSL_VERSION_NUMBER 0xnnnnnnnnnL
#include <openssl/crypto.h>

long SSLeay(
        void
);

char *SSLeay_version(
        int t
);
```

DESCRIPTION

OPENSSL_VERSION_NUMBER is a numeric release version identifier:

`MMNNFFPPS: major minor fix patch status`

The status nibble has one of the values 0 for development, 1 to e for betas 1 to 14, and f for release.

for example

```
0x000906000 == 0.9.6 dev
0x000906023 == 0.9.6b beta 3
0x00090605f == 0.9.6e release
```

Versions prior to 0.9.3 have identifiers < 0x0930. Versions between 0.9.3 and 0.9.5 had a version identifier with this interpretation:

```
MMNNFFRBB major minor fix final beta/patch
```

Examples are:

```
0x000904100 == 0.9.4 release
0x000905000 == 0.9.5 dev
```

Version 0.9.5a had an interim interpretation that is like the current one, except the patch level got the highest bit set, to keep continuity. The number was therefore 0x0090581f.

For backward compatibility, SSLEAY_VERSION_NUMBER is also defined.

The `SSLeay()` function returns this number. The return value can be compared to the macro to make sure that the correct version of the library has been loaded, especially when using DLLs on Windows systems.

The `SSLeay_version()` function returns different strings depending on `t`:

- SSLEAY_VERSION

  The text variant of the version number and the release date. For example, `OpenSSL 0.9.5a 1 Apr 2000`

- SSLEAY_CFLAGS

  The flags given to the C compiler when compiling OpenSSL are returned in a string.

- SSLEAY_PLATFORM

The platform name used when OpenSSL was configured is returned.

If the data request isn't available, text saying that the information is not available is returned.

For an unknown `t`, the text `not available` is returned.

RETURN VALUE

The version number.

HISTORY

The `SSLeay()` and `SSLEAY_VERSION_NUMBER()` functions are available in all versions of SSLeay and OpenSSL. The `OPENSSL_VERSION_NUMBER()` is available in all versions of OpenSSL.

SEE ALSO

Functions: *crypto*

passwd

NAME

passwd – Compute password hashes

SYNOPSIS

openssl passwd [-crypt] [-1] [-apr1] [-salt *string*] [-in *file*] [-stdin] [-quiet] [-table] [-*password*]

OPTIONS

crypt

Uses the crypt algorithm (default).

1

Uses the MD5 based BSD password algorithm 1.

apr1

Uses the apr1 algorithm (Apache variant of the BSD algorithm).

salt *string*

Uses the specified salt.

in *file*

Reads passwords from *file*.

stdin

Reads passwords from stdin.

quiet

Does not output warnings when passwords given at the command line are truncated.

table

Prepends the cleartext password and a TAB character to each password hash in the output list.

DESCRIPTION

The passwd command computes the hash of a password typed at run-time or the hash of each password in a list. The password list is taken from the named file for option infile, from stdin for option stdin, and from the command line. The UNIX standard algorithm crypt and the MD5-based BSD password algorithm 1 and its Apache variant apr1 are available.

EXAMPLES

openssl passwd -crypt -salt xx password

Prints xxj31ZMTZzkVA.

openssl passwd -1 -salt xxxxxxxx password

Prints 1xxxxxxxx$8XJIcl6ZXqBMCK0qFevqT1.

```
openssl passwd -apr1 -salt xxxxxxxx password
```

Prints `$apr1$xxxxxxxx$dxHfLAsjHkDRmG83UXe8K0`.

pem

NAME

pem – PEM routines

SYNOPSIS

```
#include <openssl/pem.h>
EVP_PKEY *PEM_read_bio_PrivateKey(
        BIO *bp)
        (EVP_PKEY **x)
        (pem_password_cb *cb)
        (void *u
);
EVP_PKEY *PEM_read_PrivateKey(
        FILE *fp)
        (EVP_PKEY **x)
        (pem_password_cb *cb)
        (void *u
);
int PEM_write_bio_PrivateKey(
        BIO *bp)
        (EVP_PKEY *x)
        (const EVP_CIPHER *enc)
        (unsigned char *kstr)
        (int klen)
        (pem_password_cb *cb)
        (void *u
);
int PEM_write_PrivateKey(
        FILE *fp)
        (EVP_PKEY *x)
        (const EVP_CIPHER *enc)
        (unsigned char *kstr)
        (int klen)
        (pem_password_cb *cb)
        (void *u
);
int PEM_write_bio_PKCS8PrivateKey(
        BIO *bp)
        (EVP_PKEY *x)
        (const EVP_CIPHER *enc)
        (char *kstr)
```

```
        (int klen)
        (pem_password_cb *cb)
        (void *u
);
int PEM_write_PKCS8PrivateKey(
        FILE *fp)
        (EVP_PKEY *x)
        (const EVP_CIPHER *enc)
        (char *kstr)
        (int klen)
        (pem_password_cb *cb)
        (void *u
);
int PEM_write_bio_PKCS8PrivateKey_nid(
        BIO *bp)
        (EVP_PKEY *x)
        (int nid)
        (char *kstr)
        (int klen)
        (pem_password_cb *cb)
        (void *u
);
int PEM_write_PKCS8PrivateKey_nid(
        FILE *fp)
        (EVP_PKEY *x)
        (int nid)
        (char *kstr)
        (int klen)
        (pem_password_cb *cb)
        (void *u
);
EVP_PKEY *PEM_read_bio_PUBKEY(
        BIO *bp)
        (EVP_PKEY **x)
        (pem_password_cb *cb)
        (void *u
);
EVP_PKEY *PEM_read_PUBKEY(
        FILE *fp)
        (EVP_PKEY **x)
        (pem_password_cb *cb)
        (void *u
```

```
);
int PEM_write_bio_PUBKEY(
        BIO *bp)
        (EVP_PKEY *x
);
int PEM_write_PUBKEY(
        FILE *fp)
        (EVP_PKEY *x
);
RSA *PEM_read_bio_RSAPrivateKey(
        BIO *bp)
        (RSA **x)
        (pem_password_cb *cb)
        (void *u
);
RSA *PEM_read_RSAPrivateKey(
        FILE *fp)
        (RSA **x)
        (pem_password_cb *cb)
        (void *u
);
int PEM_write_bio_RSAPrivateKey(
        BIO *bp)
        (RSA *x)
        (const EVP_CIPHER *enc)
        (int klen)
        (unsigned char *kstr)
        (pem_password_cb *cb)
        (void *u
);
int PEM_write_RSAPrivateKey(
        FILE *fp)
        (RSA *x)
        (const EVP_CIPHER *enc)
        (unsigned char *kstr)
        (int klen)
        (pem_password_cb *cb)
        (void *u
);
RSA *PEM_read_bio_RSAPublicKey(
        BIO *bp)
```

```
        (RSA **x)
        (pem_password_cb *cb)
        (void *u
);
RSA *PEM_read_RSAPublicKey(
        FILE *fp)
        (RSA **x)
        (pem_password_cb *cb)
        (void *u
);
int PEM_write_bio_RSAPublicKey(
        BIO *bp)
        (RSA *x
);
int PEM_write_RSAPublicKey(
        FILE *fp)
        (RSA *x
);
RSA *PEM_read_bio_RSA_PUBKEY(
        BIO *bp)
        (RSA **x)
        (pem_password_cb *cb)
        (void *u
);
RSA *PEM_read_RSA_PUBKEY(
        FILE *fp)
        (RSA **x)
        (pem_password_cb *cb)
        (void *u
);
int PEM_write_bio_RSA_PUBKEY(
        BIO *bp)
        (RSA *x
);
int PEM_write_RSAPublicKey(
        FILE *fp)
        (RSA *x
);
RSA *PEM_read_bio_RSA_PUBKEY(
        BIO *bp)
        (RSA **x)
```

```
        (pem_password_cb *cb)
        (void *u
);
RSA *PEM_read_RSA_PUBKEY(
        FILE *fp)
        (RSA **x)
        (pem_password_cb *cb)
        (void *u
);
int PEM_write_bio_RSA_PUBKEY(
        BIO *bp)
        (RSA *x
);
int PEM_write_RSA_PUBKEY(
        FILE *fp)
        (RSA *x
);
DSA *PEM_read_bio_DSAPrivateKey(
        BIO *bp)
        (DSA **x)
        (pem_password_cb *cb)
        (void *u
);
DSA *PEM_read_DSAPrivateKey(
        FILE *fp)
        (DSA **x)
        (pem_password_cb *cb)
        (void *u
);
int PEM_write_bio_DSAPrivateKey(
        BIO *bp)
        (DSA *x, const EVP_CIPHER *enc)
        (unsigned char *kstr)
        (int klen)
        (pem_password_cb *cb)
        (void *u
);
int PEM_write_DSAPrivateKey(
        FILE *fp)
        (DSA *x)
        (const EVP_CIPHER *enc)
```

```
        (unsigned char *kstr)
        (int klen)
        (pem_password_cb *cb)
        (void *u
);
DSA *PEM_read_bio_DSA_PUBKEY(
        BIO *bp)
        (DSA **x)
        (pem_password_cb *cb)
        (void *u
);
DSA *PEM_read_DSA_PUBKEY(
        FILE *fp)
        (DSA **x)
        (pem_password_cb *cb)
        (void *u
);
int PEM_write_bio_DSA_PUBKEY(
        BIO *bp)
        (DSA *x
);
int PEM_write_DSA_PUBKEY(
        FILE *fp)
        (DSA *x
);
DSA *PEM_read_bio_DSAparams(
        BIO *bp)
        (DSA **x)
        (pem_password_cb *cb)
        (void *u
);
DSA *PEM_read_DSAparams(
        FILE *fp)
        (DSA **x)
        (pem_password_cb *cb)
        (void *u
);
int PEM_write_bio_DSAparams(
        BIO *bp)
        (DSA *x
);
```

```
int PEM_write_DSAparams(
        FILE *fp)
        (DSA *x
);
DH *PEM_read_bio_DHparams(
        BIO *bp)
        (DH **x)
        (pem_password_cb *cb)
        (void *u
);
DH *PEM_read_DHparams(
        FILE *fp)
        (DH **x)
        (pem_password_cb *cb)
        (void *u
);
int PEM_write_bio_DHparams(
        BIO *bp)
        (DH *x
);
int PEM_write_DHparams(
        FILE *fp)
        (DH *x
);
X509_CRL *PEM_read_bio_X509_CRL(
        BIO *bp)
        (X509_CRL **x)
        (pem_password_cb *cb)
        (void *u
);
X509_CRL *PEM_read_X509_CRL(
        FILE *fp)
        (X509_CRL **x)
        (pem_password_cb *cb)
        (void *u
);
int PEM_write_bio_X509_CRL(
        BIO *bp)
        (X509_CRL *x
);
int PEM_write_X509_CRL(
```

```
        FILE *fp)
        (X509_CRL *x
);
PKCS7 *PEM_read_bio_PKCS7(
        BIO *bp)
        (PKCS7 **x)
        (pem_password_cb *cb)
        (void *u
);
PKCS7 *PEM_read_PKCS7(
        FILE *fp)
        (PKCS7 **x)
        (pem_password_cb *cb)
        (void *u
);
int PEM_write_bio_PKCS7(
        BIO *bp)
        (PKCS7 *x
);
int PEM_write_PKCS7(
        FILE *fp)
        (PKCS7 *x
);
NETSCAPE_CERT_SEQUENCE *PEM_read_bio_NETSCAPE_CERT_SEQUENCE(
        BIO *bp)
        (NETSCAPE_CERT_SEQUENCE **x)
        (pem_password_cb *cb)
        (void *u
);
NETSCAPE_CERT_SEQUENCE *PEM_read_NETSCAPE_CERT_SEQUENCE(
        FILE *fp)
        (NETSCAPE_CERT_SEQUENCE **x)
        (pem_password_cb *cb)
        (void *u
);
int PEM_write_bio_NETSCAPE_CERT_SEQUENCE(
        BIO *bp)
        (NETSCAPE_CERT_SEQUENCE *x
);
int PEM_write_NETSCAPE_CERT_SEQUENCE(
        FILE *fp)
```

```
        (NETSCAPE_CERT_SEQUENCE *x
    );
```

DESCRIPTION

The `pem()` functions read or write structures in PEM format. In this sense PEM format is simply base64 encoded data surrounded by header lines.

Each operation has four functions associated with it. For clarity the term foobar functions will be used to collectively refer to the `PEM_read_bio_foobar()`, `PEM_read_foobar()`, `PEM_write_bio_foobar()`, and `PEM_write_foobar()` functions.

The PrivateKey functions read or write a private key in PEM format using an EVP_PKEY structure. The write routines use traditional private key format and can handle both RSA and DSA private keys. The read functions can transparently handle PKCS#8 format encrypted and unencrypted keys too.

The `PEM_write_bio_PKCS8PrivateKey()` and `PEM_write_PKCS8PrivateKey()` functions write a private key in an EVP_PKEY structure in PKCS#8 EncryptedPrivateKeyInfo format using PKCS#5 v2.0 password based encryption algorithms. The cipher argument specifies the encryption algoritm to use. Unlike all other PEM routines the encryption is applied at the PKCS#8 level and not in the PEM headers. If cipher is NULL then no encryption is used and a PKCS#8 PrivateKeyInfo structure is used instead.

The `PEM_write_bio_PKCS8PrivateKey_nid()` and `PEM_write_PKCS8PrivateKey_nid()` functions also write out a private key as a PKCS#8 EncryptedPrivateKeyInfo however it uses PKCS#5 v1.5 or PKCS#12 encryption algorithms instead. The algorithm to use is specified in the nid parameter and should be the NID of the corresponding OBJECT IDENTIFIER (see Notes section).

The PUBKEY functions process a public key using an EVP_PKEY structure. The public key is encoded as a SubjectPublicKeyInfo structure.

The RSAPrivateKey functions process an RSA private key using an RSA structure. It handles the same formats as the PrivateKey functions but an error occurs if the private key is not RSA.

The RSAPublicKey functions process an RSA public key using an RSA structure. The public key is encoded using a PKCS#1 RSAPublicKey structure.

The RSA_PUBKEY functions also process an RSA public key using an RSA structure. However the public key is encoded using a SubjectPublicKeyInfo structure and an error occurs if the public key is not RSA.

The DSAPrivateKey functions process a DSA private key using a DSA structure. It handles the same formats as the PrivateKey functions but an error occurs if the private key is not DSA.

The DSA_PUBKEY functions process a DSA public key using a DSA structure. The public key is encoded using a SubjectPublicKeyInfo structure and an error occurs if the public key is not DSA.

The DSAparams functions process DSA parameters using a DSA structure. The parameters are encoded using a foobar structure.

The DHparams functions process DH parameters using a DH structure. The parameters are encoded using a PKCS#3 DHparameter structure.

The X509 functions process an X509 certificate using an X509 structure. They will also process a trusted X509 certificate but any trust settings are discarded.

The X509_AUX functions process a trusted X509 certificate using an X509 structure.

The X509_REQ and X509_REQ_NEW functions process a PKCS#10 certificate request using an X509_REQ structure. The X509_REQ write functions use CERTIFICATE REQUEST in the header whereas the X509_REQ_NEW functions use NEW CERTIFICATE REQUEST (as required by some CAs). The X509_REQ read functions will handle either form so there are no X509_REQ_NEW read functions.

The X509_CRL functions process an X509 CRL using an X509_CRL structure.

The PKCS7 functions process a PKCS#7 ContentInfo using a PKCS7 structure.

The NETSCAPE_CERT_SEQUENCE functions process a Netscape Certificate Sequence using a NETSCAPE_CERT_SEQUENCE structure.

PEM FUNCTION ARGUMENTS

The PEM functions have many common arguments.

The *bp* IO parameter (if present) specifies the BIO to read from or write to.

The *fp* FILE parameter (if present) specifies the FILE pointer to read from or write to.

The PEM read functions all take an argument `TYPE **x` and return a `TYPE *` pointer. Where TYPE is whatever structure the function uses. If x is NULL then the parameter is ignored. If x is not NULL but **x* is NULL then the structure returned will be written to **x*. If neither x nor *x is NULL then an attempt is made to reuse the structure at **x* (see NOTES and EXAMPLES sections). Irrespective of the value of x, a pointer to the structure is always returned (or NULL if an error occurred).

The PEM functions which write private keys take an *enc* parameter which specifies the encryption algorithm to use. Encryption is done at the PEM level. If this parameter is set to NULL then the private key is written in unencrypted form.

The *cb* argument is the callback to use when querying for the passphrase used for encrypted PEM structures (normally only private keys).

For the PEM write routines if the *kstr* parameter is not NULL then *klen* bytes at *kstr* are used as the passphrase and *cb* is ignored.

If the *cb* parameter is set to NULL and the *u* parameter is not NULL then the *u* parameter is interpreted as a null terminated string to use as the passphrase. If both *cb* and *u* are NULL then the default callback routine is used which will typically prompt for the passphrase on the current terminal with echoing turned off.

The default passphrase callback is sometimes inappropriate (for example in a GUI application) so an alternative can be supplied. The callback routine has the following form: int `cb(char *buf, int size, int rwflag, void *u)`; `buf` is the buffer to write the passphrase to. Size is the maximum length of the passphrase (i.e. the size of buf). `rwflag` is a flag which is set to 0 when reading and 1 when writing. A typical routine will ask the user to verify the passphrase (for example by prompting for it twice) if `rwflag` is 1. The *u* parameter has the same value as the *u* parameter passed to the PEM routine. It allows arbitrary data to be passed to the callback by the application (for example, a window handle in a GUI application). The callback must return the number of characters in the passphrase or 0 if an error occurred.

NOTES

The PEM read routines in some versions of OpenSSL will not correctly reuse an existing structure. Therefore the following may not work where x already contains a valid certificate:

```
PEM_read_bio(bp, &x, 0, NULL);
```

However, the following is guaranteed to work:

```
X509_free(x);
x =3D PEM_read_bio(bp, NULL, 0, NULL);
```

The old PrivateKey write routines are retained for compatibility. New applications should write private keys using the `PEM_write_bio_PKCS8PrivateKey()` or `PEM_write_PKCS8PrivateKey()` routines because they are more secure, unless compatibility with older versions of OpenSSL is important. (They use an iteration

count of 2048, whereas the traditional routines use a count of 1.) The PrivateKey read routines can be used in all applications because they handle all formats transparently. A frequent cause of problems is attempting to use the PEM routines in the following manner:

```
X509 *x;
PEM_read_bio_X509(bp, &x, 0, NULL);
```

This is a bug because an attempt will be made to reuse the data at x which is an uninitialized pointer.

RETURN VALUES

The read routines return either a pointer to the structure read or NULL is an error occurred.

The write routines return 1 for success or 0 for failure.

EXAMPLES

Although the PEM routines take several arguments in almost all applications most of them are set to 0 or NULL.

Read a certificate in PEM format from a BIO:

```
X509 *x;
x = PEM_read_bio(bp, NULL, 0, NULL);
if (x == NULL)
    {

    }
```

Alternative method:

```
X509 *x = NULL;
if (!PEM_read_bio_X509(bp, &x, 0, NULL))
       {

       }
```

Write a certificate to a BIO:

```
if (!PEM_write_bio_X509(bp, x))
       {

       }
```

Write an unencrypted private key to a FILE pointer:

```
if (!PEM_write_PrivateKey(fp, key, NULL, NULL, 0, 0, NULL))
       {

       }
```

Write a private key (using traditional format) to a BIO using triple DES encryption, the passphrase is prompted for:

```
if (!PEM_write_bio_PrivateKey(bp, key, EVP_des_ede3_cbc(), NULL, 0, 0, NULL))
       {

       }
```

Write a private key (using PKCS#8 format) to a BIO using triple DES encryption, using the passphrase "hello":

```
if (!PEM_write_bio_PKCS8PrivateKey(bp, key, EVP_des_ede3_cbc(),
                                   NULL, 0, 0, "hello"))
       {

       }
```

Read a private key from a BIO using the passphrase "hello":

```
key = PEM_read_bio_PrivateKey(bp, NULL, 0, "hello");
if (key == NULL)
        {

        }
```

Read a private key from a BIO using a passphrase callback:

```
key = PEM_read_bio_PrivateKey(bp, NULL, pass_cb, "My Private Key");
if (key == NULL)
        {

        }
```

Skeleton passphrase callback:

```
int pass_cb(char *buf, int size, int rwflag, void *u);
       {
       int len;
       char *tmp;

       printf("Enter passphrase for \"%s\"\n", u);

       tmp = "hello";
       len = strlen(tmp);

       if (len <= 0) return 0;

       if (len > size) len = size;
       memcpy(buf, tmp, len);
       return len;
       }
```

pkcs12

NAME

pkcs12 – PKCS#12 file utility

SYNOPSIS

openssl pkcs12 [-export] [-chain] [-inkey *filename*] [-certfile *filename*] [name *name*] [-caname *name*] [-in *filename*] [-out *filename*] [-noout] [-nomacver] [-nocerts] [-clcerts] [-cacerts] [-nokeys] [-info] [-des] [-des3] [-idea] [-nodes] [-noiter] [-maciter] [-twopass] [-descert] [-certpbe] [-keypbe] [-keyex] [-keysig] [-password *arg*] [-passin *arg*] [-passout *arg*] [-rand *filename*]

OPTIONS

There are many options. The meaning of some depends on whether a PKCS#12 file is being created or parsed. By default a PKCS#12 file is parsed. A PKCS#12 file can be created by using the `export` option.

PARSING OPTIONS

in *filename*

This specifies filename of the PKCS#12 file to be parsed. Standard input is used by default.

out *filename*

The filename to write certificates and private keys to, standard output by default. They are all written in PEM format.

pass *arg*, passin *arg*

The PKCS#12 file (i.e. input file) password source. For more information about the format of *arg*, see the Pass Phrase Arguments section in *openssl*.

passout *arg*

The pass phrase source to encrypt any outputed private keys with. For more information about the format of **arg** see the Pass Phrase Arguments section in *openssl*.

noout

Inhibits output of the keys and certificates to the output file version of the PKCS#12 file.

clcerts

Only output client certificates (not CA certificates).

cacerts

Only output CA certificates (not client certificates).

nocerts

No certificates will be output.

nokeys

No private keys will be output.

info

Outputs additional information about the PKCS#12 file structure, algorithms used and iteration counts.

`des`

Uses DES to encrypt private keys before outputting.

`des3`

Uses triple DES to encrypt private keys before outputting, this is the default.

`idea`

Uses IDEA to encrypt private keys before outputting.

`nodes`

Does not encrypt the private keys.

`nomacver`

Does not attempt to verify the integrity MAC before reading the file.

`twopass`

Prompts for separate integrity and encryption passwords. Most software always assumes these are the same so this option will render such PKCS#12 files unreadable.

FILE CREATION OPTIONS

`export`

Specifies that a PKCS#12 file will be created rather than parsed.

`out` *filename*

Specifies the filename where the PKCS#12 file is written. Standard output is used by default.

`in` *filename*

The filename to read certificates and private keys from, standard input by default. They must all be in PEM format. The order does not matter, but one private key and its corresponding certificate should be present. If additional certificates are present they will also be included in the PKCS#12 file.

`inkey` *filename*

The file to read private key from. If not present then a private key must be present in the input file.

`name` *friendlyname*

Specifies the -friendly name- for the certificate and private key. This name is typically displayed in list boxes by software importing the file.

`certfile` *filename*

A filename to read additional certificates from.

`caname` *friendlyname*

Specifies the -friendly name- for other certificates. This option may be used multiple times to specify names for all certificates in the order they appear. Netscape ignores friendly names on other certificates whereas MSIE displays them.

`pass` *arg*, `passout` *arg*

The PKCS#12 file (i.e. output file) password source. For more information about the format of *arg*, see the Pass Phase Arguments section in *openssl*.

`passin` *password*

Pass phrase source used to decrypt any input private keys. For more information about the format of *arg*, see the Pass Phrase Arguments section in *openssl*.

`chain`

If this option is present then an attempt is made to include the entire certificate chain of the user certificate. The standard CA store is used for this search. If the search fails it is considered a fatal error.

`descert`

Encrypts the certificate using triple DES. This may render the PKCS#12 file unreadable by some export grade software. By default the private key is encrypted using triple DES and the certificate using 40 bit RC2.

`keypbe` *alg*, `certpbe` *alg*

Allows the algorithm used to encrypt the private key and certificates to be selected. Although any PKCS#5 v1.5 or PKCS#12 algorithms can be selected, it is advisable only to use PKCS#12 algorithms. See the list in the Notes section for more information.

`keyex|keysig`

Specifies that the private key is to be used for key exchange or just signing. This option is only interpreted by MSIE and similar MS software. Normally export grade software will only allow 512 bit RSA keys to be used for encryption purposes but arbitrary length keys for signing. The `keysig` option marks the key for signing only. Signing only keys can be used for S/MIME signing, authenticode (ActiveX control signing) and SSL client authentication, however due to a bug only MSIE 5.0 and later support the use of signing only keys for SSL client authentication.

`nomaciter`, `noiter`

These options affect the iteration counts on the MAC and key algorithms. Unless you wish to produce files compatible with MSIE 4.0 you should leave these options alone.

To discourage attacks by using large dictionaries of common passwords the algorithm that derives keys from passwords can have an iteration count applied to it: this causes a certain part of the algorithm to be repeated and slows it down. The MAC is used to check the file integrity but since it will normally have the same password as the keys and certificates it could also be attacked. By default both MAC and encryption iteration counts are set to 2048, using these options the MAC and encryption iteration counts can be set to 1, since this reduces the file security you should not use these options unless you really have to. Most software supports both MAC and key iteration counts. MSIE 4.0 doesn't support MAC iteration counts so it needs the `nomaciter` option.

`maciter`

This option is included for compatibility with previous versions. It used to be needed to use MAC iterations counts but they are now used by default.

`rand` *filename*

A file or files containing random data used to seed the random number generator, or an EGD socket. (See *RAND_egd*.) Multiple files can be specified separated by an OS-dependent character. The separator is a semicolon (;) for MS-Windows, a comma (,) for OpenVMS, and a colon (:) for all others.

DESCRIPTION

The `pkcs12` command allows PKCS#12 files (sometimes referred to as PFX files) to be created and parsed. PKCS#12 files are used by several programs including Netscape, MSIE and MS Outlook.

NOTES

Although there are a large number of options most of them are very rarely used. For PKCS#12 file parsing only the `in` and `out` options need to be used for PKCS#12 file creation. The `export` and `name` are also used.

If none of the `clcerts`, `cacerts` or `nocerts` options are present then all certificates will be output in the order they appear in the input PKCS#12 files. There is no guarantee that the first certificate present is the one corresponding to the private key. Certain software which requires a private key and certificate and assumes the first certificate in the file is the one corresponding to the private key: this may not always be the case. Using the `clcerts` option will solve this problem by only outputing the certificate corresponding to the private key. If the CA certificates are required then they can be output to a separate file using the `nokeys`and `cacerts` options.

The `keypbe` and `certpbe` algorithms allow the precise encryption algorithms for private keys and certificates to be specified. Normally the defaults are fine, but occasionally software cannot handle triple DES encrypted private keys. In that case, the `-keypbe PBE-SHA1-RC2-40` option can be used to reduce the private key encryption to 40 bit RC2. A complete description of all algorithms is contained in the *pkcs8*reference page.

RESTRICTIONS

Versions of OpenSSL before 0.9.6a had a bug in the PKCS#12 key generation routines. Under rare circumstances this could produce a PKCS#12 file encrypted with an invalid key. As a result some PKCS#12 files which triggered this bug from other implementations (MSIE or Netscape) could not be decrypted by OpenSSL and similarly OpenSSL could produce PKCS#12 files which could not be decrypted by other implementations. The chances of producing such a file are relatively small - less than 1 in 256.

A side effect of fixing this bug is that any old invalidly encrypted PKCS#12 files can no longer be parsed by the fixed version. Under such circumstances the `pkcs12` utility will report that the MAC is OK but fail with a decryption error when extracting private keys.

This problem can be resolved by extracting the private keys and certificates from the PKCS#12 file using an older version of OpenSSL and recreating the PKCS#12 file from the keys and certificates using a newer version of OpenSSL. For example:

```
old-openssl -in bad.p12 -out keycerts.pem
 openssl -in keycerts.pem -export -name "My PKCS#12 file" -out fixed.p12
```

EXAMPLES

Parse a PKCS#12 file and output it to a file:

```
 openssl pkcs12 -in file.p12 -out file.pem
```

Output only client certificates to a file:

```
 openssl pkcs12 -in file.p12 -clcerts -out file.pem
```

Don't encrypt the private key:

```
openssl pkcs12 -in file.p12 -out file.pem -nodes
```

Print some information about a PKCS#12 file:

```
openssl pkcs12 -in file.p12 -info -noout
```

Create a PKCS#12 file:

```
openssl pkcs12 -export -in file.pem -out file.p12 -name "My Certificate"
```

Include some extra certificates:

```
openssl pkcs12 -export -in file.pem -out file.p12 -name "My Certificate" \
 -certfile othercerts.pem
```

SEE ALSO

Commands: *pkcs8*

pkcs7

NAME

pkcs7 – PKCS#7 utility

SYNOPSIS

openssl pkcs7 [-inform PEM|DER] [-outform PEM|DER] [-in *filename*] [-out *filename*] [-print_certs] [-text] [-noout]

OPTIONS

`inform DER|PEM`

Specifies the input format. The `DER` format is DER encoded PKCS#7 v1.5 structure. The `PEM` format (the default) is a base64 encoded version of the `DER` form with header and footer lines.

`outform DER|PEM`

Specifies the output format. The options have the same meaning as the `inform` option.

`in` *filename*

Specifies the input filename to read from or standard input if this option is not specified.

`out` *filename*

Specifies the output filename to write to or standard output by default.

`print_certs`

Prints out any certificates or CRLs contained in the file. They are preceded by their subject and issuer names in one-line format.

`text`

Prints out certificate details in full rather than just subject and issuer names.

`noout`

Does not output the encoded version of the PKCS#7 structure (or certificates if `print_certs` is set).

DESCRIPTION

The `pkcs7` command processes PKCS#7 files in `DER` or `PEM` format.

NOTES

The PEM PKCS#7 format uses the header and footer lines:

```
-----BEGIN PKCS7-----
-----END PKCS7-----
```

For compatability with some CAs it will also accept:

```
-----BEGIN CERTIFICATE-----
-----END CERTIFICATE-----
```

RESTRICTIONS

There is no option to print out all the fields of a PKCS#7 file.

These PKCS#7 routines only understand PKCS#7 v 1.5 as specified in RFC2315. For example, they cannot currently parse the new CMS as described in RFC2630.

EXAMPLES

Convert a PKCS#7 file from `PEM` to `DER`:

```
openssl pkcs7 -in file.pem -outform DER -out file.der
```

Output all certificates in a file:

```
openssl pkcs7 -in file.pem -print_certs -out certs.pem
```

SEE ALSO

Commands: *crl2pkcs7*

pkcs8

NAME

pkcs8 – PKCS#8 format private key conversion tool

SYNOPSIS

`openssl pkcs8 [-topk8] [-inform PEM|DER] [-outform PEM|DER] [-in` *`filename`*`] [-passin` *`arg`*`] [-out` *`filename`*`] [-passout` *`arg`*`] [-noiter] [-nocrypt] [-nooct] [-embed] [-nsdb] [-v2` *`alg`*`] [-v1` *`alg`*`]`

STANDARDS

Test vectors from this PKCS#5 v2.0 implementation were posted to the pkcs-tng mailing list using triple DES, DES and RC2 with high iteration counts. Several people confirmed that they could decrypt the private keys produced. Therefore it can be assumed that the PKCS#5 v2.0 implementation is reasonably accurate, at least as far as these algorithms are concerned.

The format of PKCS#8 DSA and other private keys is not well documented. It is hidden away in PKCS#11 v2.01, section 11.9. OpenSSL's default DSA PKCS#8 private key format complies with this standard.

OPTIONS

`topk8`

Normally a PKCS#8 private key is expected on input and a traditional format private key will be written. With the `topk8` option the situation is reversed; it reads a traditional format private key and writes a PKCS#8 format key.

`inform DER|PEM`

Specifies the input format. If a PKCS#8 format key is expected on input then either a `DER` or `PEM` encoded version of a PKCS#8 key will be expected. Otherwise the `DER` or `PEM` format of the traditional format private key is used.

`outform DER|PEM`

Specifies the output format. The options have the same meaning as the `inform` option.

`in` *`filename`*

Specifies the input filename to read a key from or standard input if this option is not specified. If the key is encrypted there is a prompt for a pass phrase.

`passin` *`arg`*

Input file password source. For more information about the format of `arg`, see the Pass Phrase Arguments section in *openssl*.

`out` *`filename`*

Specifies the output filename to write a key to or standard output by default. If any encryption options are set, there is a prompt for a pass phrase. The output filename should not be the same as the input filename.

`passout` *`arg`*

Output file password source. For more information about the format of `arg` see the Pass Phrase Arguments section in *openssl*.

`nocrypt`

PKCS#8 keys generated or input are normally PKCS#8 `EncryptedPrivateKeyInfo` structures using an appropriate password based encryption algorithm. With this option an unencrypted `PrivateKeyInfo` structure is expected or output. This option does not encrypt private keys, and should only be used when absolutely necessary. Certain software such as some versions of Java code signing software used unencrypted private keys.

`nooct`

Generates RSA private keys in a broken format used by some software. Specifically the private key should be enclosed in a octet string, but some software only includes the structure itself without the surrounding octet string.

`embed`

Generates DSA keys in a broken format. The DSA parameters are embedded inside the PrivateKey structure. In this form the octet string contains an ASN1 sequence consisting of two structures: a sequence containing the parameters and an ASN1 integer containing the private key.

`nsdb`

Generates DSA keys in a broken format compatible with Netscape private key databases. The PrivateKey contains a sequence consisting of the public and private keys respectively.

`v2` *`alg`*

Enables the use of PKCS#5 v2.0 algorithms. Normally PKCS#8 private keys are encrypted with the password based encryption algorithm called `pbeWithMD5AndDES-CBC`. This uses 56-bit DES encryption, but it was the strongest encryption algorithm supported in PKCS#5 v1.5. Using the `v2` option PKCS#5 v2.0 algorithms are used which can use any encryption algorithm such as 168-bit triple DES or 128-bit RC2. However, not many implementations support PKCS#5 v2.0. If you are using private keys only with OpenSSL then this doesn't matter.

The `alg` argument is the encryption algorithm to use. Valid values include `des`, `des3` and `rc2`. We recommend that `des3` be used.

`v1` *`alg`*

Specifies a PKCS#5 v1.5 or PKCS#12 algorithm to use. A complete list of possible algorithms is included below.

PKCS#5 v1.5 and PKCS#12 algorithms.

Various algorithms can be used with the `v1` command line option, including PKCS#5 v1.5 and PKCS#12. These are described in more detail below.

`PBE-MD2-DES PBE-MD5-DES`

These algorithms were included in the original PKCS#5 v1.5 specification. They only offer 56 bits of protection since they both use `DES`.

`PBE-SHA1-RC2-64 PBE-MD2-RC2-64 PBE-MD5-RC2-64 PBE-SHA1-DES`

These algorithms are not mentioned in the original PKCS#5 v1.5 specification, but they use the same key derivation algorithm and are supported by some software. They are mentioned in PKCS#5 v2.0. They use either 64-bit RC2 or 56-bit `DES`.

`PBE-SHA1-RC4-128 PBE-SHA1-RC4-40 PBE-SHA1-3DES PBE-SHA1-2DES PBE-SHA1-RC2-128 PBE-SHA1-RC2-40`

These algorithms use the PKCS#12 password based encryption algorithm and allow strong encryption algorithms like triple DES or 128-bit RC2 to be used.

DESCRIPTION

The pkcs8 command processes private keys in PKCS#8 format. It can handle both unencrypted PKCS#8 PrivateKeyInfo format and EncryptedPrivateKeyInfo format with a variety of PKCS#5 (v1.5 and v2.0) and PKCS#12 algorithms.

NOTES

The encrypted form of PEM encoded PKCS#8 files uses the following headers and footers:

```
-----BEGIN ENCRYPTED PRIVATE KEY-----
-----END ENCRYPTED PRIVATE KEY-----
```

The unencrypted form uses:

```
-----BEGIN PRIVATE KEY-----
-----END PRIVATE KEY-----
```

Private keys encrypted using PKCS#5 v2.0 algorithms and high iteration counts are more secure than those encrypted using the traditional SSLeay compatible formats. If additional security is important, the keys should be converted. The default encryption is only 56 bits because this is the encryption that most current implementations of PKCS#8 will support.

Some software may use PKCS#12 password based encryption algorithms with PKCS#8 format private keys. These are handled automatically, but there is no option to produce them.

It is possible to write out DER encoded encrypted private keys in PKCS#8 format because the encryption details are included at an ASN1 level, whereas the traditional format includes them at a PEM level.

RESTRICTIONS

PKCS#8 using triple DES and PKCS#5 v2.0 should be the default private key format for OpenSSL. For compatibility, several of the utilities use the old format.

EXAMPLES

Convert a private from traditional to PKCS#5 v2.0 format using triple DES:

```
openssl pkcs8 -in key.pem -topk8 -v2 des3 -out enckey.pem
```

Convert a private key to PKCS#8 using a PKCS#5 1.5 compatible algorithm (DES):

```
openssl pkcs8 -in key.pem -topk8 -out enckey.pem
```

Convert a private key to PKCS#8 using a PKCS#12 compatible algorithm (3DES):

```
openssl pkcs8 -in key.pem -topk8 -out enckey.pem -v1 PBE-SHA1-3DES
```

Read a DER unencrypted PKCS#8 format private key:

```
openssl pkcs8 -inform DER -nocrypt -in key.der -out key.pem
```

Convert a private key from any PKCS#8 format to traditional format:

```
openssl pkcs8 -in pk8.pem -out key.pem
```

rand

NAME

rand – Generate pseudo-random bytes

SYNOPSIS

`openssl rand [-out filename] [-rand filename] [-base64 num]`

OPTIONS

`out` *`file`*

Writes to *`file`* instead of standard output.

`rand` *`file(s)`*

Uses a specified file or files or EGD socket for seeding the random number generator. (See *RAND_egd*.) Multiple files can be separated by an OS-dependent character. The separator is a semicolon (:) for MS-Windows, a comma (,) for OpenVMS, and a colon (:) for all others.

`base64`

Performs base64 encoding on the output.

DESCRIPTION

The `rand` command outputs *num* pseudo-random bytes after seeding the random number generater once. As in other `openssl` command line tools, PRNG seeding uses the file `$HOME/.rnd` or `.rnd` in addition to the files given in the `rand` option. A new `$HOME/.rnd` or `.rnd` file will be written back if enough seeding was obtained from these sources.

SEE ALSO

Functions: *RAND_bytes*

RAND_add

NAME

RAND_add, RAND_seed, RAND_status, RAND_event, RAND_screen – Add entropy to the PRNG

SYNOPSIS

```
#include <openssl/rand.h>
void RAND_seed(
        const void *buf, int num
);
void RAND_add(
        const void *buf, int num, double entropy
);
int RAND_status(
        void
);
int RAND_event(
        UINT iMsg, WPARAM wParam, LPARAM lParam
);
void RAND_screen(
        void
);
```

DESCRIPTION

The `RAND_add()` function mixes the `num` bytes at `buf` into the PRNG state. Thus, if the data at `buf` are unpredictable to an adversary, this increases the uncertainty about the state and makes the PRNG output less predictable. Suitable input comes from user interaction (random key presses, mouse movements) and certain hardware events. The `entropy` argument is (the lower bound of) an estimate of how much randomness is contained in `buf`, measured in bytes. Details about sources of randomness and how to estimate their entropy can be found in the literature, e.g. RFC 1750.

The `RAND_add()` function may be called with sensitive data such as user entered passwords. The seed values cannot be recovered from the PRNG output.

OpenSSL makes sure that the PRNG state is unique for each thread. On systems that provide `/dev/urandom`, the randomness device is used to seed the PRNG transparently. However, on all other systems, the application is responsible for seeding the PRNG by calling the `RAND_add()`, `RAND_egd()` or `RAND_load_file()` functions.

The `RAND_seed()` function is equivalent to the `RAND_add()` function when `num == entropy`.

The `RAND_event()` function collects the entropy from Windows events such as mouse movements and other user interaction. It should be called with the `iMsg`, `wParam` and `lParam` arguments of *`all`* messages sent to the window procedure. It will estimate the entropy contained in the event message (if any), and add it to the PRNG. The program can then process the messages as usual.

The `RAND_screen()` function is available for the convenience of Windows programmers. It adds the current contents of the screen to the PRNG. For applications that can catch Windows events, seeding the PRNG by calling `RAND_event()` is a significantly better source of randomness. It should be noted that both methods cannot be used on servers that run without user interaction.

RETURN VALUES

The `RAND_status()` and `RAND_event()` functions return 1 if the PRNG has been seeded with enough data, 0 otherwise.

The other functions do not return values.

HISTORY

The `RAND_seed()` and `RAND_screen()` functions are available in all versions of SSLeay and OpenSSL. The `RAND_add()` and `RAND_status()` functions have been added in OpenSSL 0.9.5, and `RAND_event()` in OpenSSL 0.9.5a.

SEE ALSO

Functions: *rand*, *RAND_egd*, *RAND_load_file*, *RAND_cleanup*

RAND_bytes

NAME

RAND_bytes, RAND_pseudo_bytes – Generate random data

SYNOPSIS

```
#include <openssl/rand.h>
int RAND_bytes(
        unsigned char *buf, int num
);
int RAND_pseudo_bytes(
        unsigned char *buf, int num
);
```

DESCRIPTION

The `RAND_bytes()` function puts `num` cryptographically strong pseudo-random bytes into `buf`. An error occurs if the PRNG has not been seeded with enough randomness to ensure an unpredictable byte sequence.

The `RAND_pseudo_bytes()` function puts `num` pseudo-random bytes into `buf`. Pseudo-random byte sequences generated by the `RAND_pseudo_bytes()` function will be unique if they are of sufficient length, but are not necessarily unpredictable. They can be used for non-cryptographic purposes and for certain purposes in cryptographic protocols, but usually not for key generation etc.

RETURN VALUES

The `RAND_bytes()` functon returns 1 on success, 0 otherwise. The error code can be obtained by using `ERR_get_error()`. The `RAND_pseudo_bytes()` function returns 1 if the bytes generated are cryptographically strong, 0 otherwise. Both functions return -1 if they are not supported by the current RAND method.

HISTORY

The `RAND_bytes()` function is available in all versions of SSLeay and OpenSSL. It has a return value since OpenSSL 0.9.5. The `RAND_pseudo_bytes()` function was added in OpenSSL 0.9.5.

SEE ALSO

Functions: *rand*, *err*, *RAND_add*

RAND_cleanup

NAME

RAND_cleanup – Erase the PRNG state

SYNOPSIS

```
#include <openssl/rand.h>
void RAND_cleanup(
        void
);
```

DESCRIPTION

The `RAND_cleanup()` function erases the memory used by the PRNG.

RETURN VALUE

The `RAND_cleanup()` function returns no value.

HISTORY

The `RAND_cleanup()` function is available in all versions of SSLeay and OpenSSL.

SEE ALSO

Functions: *rand*

RAND_egd

NAME

RAND_egd – Query entropy gathering daemon

SYNOPSIS

```
#include <openssl/rand.h>
int RAND_egd(
        const char *path
);
int RAND_egd_bytes(
        const char *path, int bytes
);
```

DESCRIPTION

The `RAND_egd()` function queries the entropy gathering daemon EGD on socket `path`. It queries 255 bytes and uses `RAND_add()` to seed the OpenSSL built-in PRNG. The `RAND_egd(path)` is a wrapper for `RAND_egd_bytes(path, 255)`.

The `RAND_egd_bytes()` function queries the entropy gathering daemon EGD on socket `path`. It queries `bytes` and uses `RAND_add()` to seed the OpenSSL built-in PRNG. This function is more flexible than the `RAND_egd()` function . When only one secret key must be generated, it is not necessary to request the full amount 255 bytes from the EGD socket. This can be advantageous, since the amount of entropy that can be retrieved from EGD over time is limited.

NOTES

On systems without `/dev/*random` devices providing entropy from the kernel, the EGD entropy gathering daemon can be used to collect entropy. It provides a socket interface through which entropy can be gathered in chunks up to 255 bytes. Several chunks can be queried during one connection.

EGD is available from http://www.lothar.com/tech/crypto/ (`perl Makefile.PL; make; make install` to install). It is run as `egd` *`path`*, where *`path`* is an absolute path designating a socket. When the `RAND_egd()` function is called with that path as an argument, it tries to read random bytes that EGD collected. The read is performed in non-blocking mode.

Alternatively, the EGD-interface compatible daemon PRNGD can be used. It is available from http://www.aet.tu-cottbus.de/personen/jaenicke/postfix_tls/prngd.html. PRNGD does employ an internal PRNG itself and can therefore never run out of entropy.

RETURN VALUE

The `RAND_egd()` and `RAND_egd_bytes()` functions return the number of bytes read from the daemon on success, and -1 if the connection failed or the daemon did not return enough data to fully seed the PRNG.

HISTORY

The `RAND_egd()` function is available since OpenSSL 0.9.5.

The `RAND_egd_bytes()` function is available since OpenSSL 0.9.6.

RAND_load_file

NAME

RAND_load_file, RAND_write_file, RAND_file_name – PRNG seed file

SYNOPSIS

```
#include <openssl/rand.h>

const char *RAND_file_name(
        char *buf, size_t num
);

int RAND_load_file(
        const char *filename, long max_bytes
);

int RAND_write_file(
        const char *filename
);
```

DESCRIPTION

The `RAND_load_file()` function reads a number of bytes from file `filename` and adds them to the PRNG. If `max_bytes` is non-negative, up to `max_bytes` are read; starting with OpenSSL 0.9.5, if `max_bytes` is -1, the complete file is read.

The `RAND_write_file()` function writes a number of random bytes (currently 1024) to file *`filename`* which can be used to initialize the PRNG by calling `RAND_load_file()` in a later session.

The `RAND_file_name()` function generates a default path for the random seed file. `buf` points to a buffer of size `num` in which to store the filename. The seed file is `$RANDFILE` if that environment variable is set, `$HOME/.rnd` otherwise. If `$HOME` is not set either, or `num` is too small for the path name, an error occurs.

RETURN VALUES

The `RAND_load_file()` function returns the number of bytes read.

The `RAND_write_file()` function returns the number of bytes written, and -1 if the bytes written were generated without appropriate seed.

The `RAND_file_name()` function returns a pointer to `buf` on success, and NULL on error.

HISTORY

The `RAND_load_file()`, `RAND_write_file()`, and `RAND_file_name()` functions are available in all versions of SSLeay and OpenSSL.

SEE ALSO

Functions: *rand*, *RAND_add*, *RAND_cleanup*

RAND_set_rand_method

NAME

RAND_set_rand_method, RAND_get_rand_method, RAND_SSLeay – Select RAND method

SYNOPSIS

```
#include <openssl/rand.h>

void RAND_set_rand_method(
        RAND_METHOD *meth
);

RAND_METHOD *RAND_get_rand_method(
       void
);

RAND_METHOD *RAND_SSLeay(
        void
);
```

DESCRIPTION

A RAND_METHOD specifies the functions that OpenSSL uses for random number generation. By modifying the method, alternative implementations such as hardware RNGs may be used. Initially, the default is to use the OpenSSL internal implementation. The RAND_SSLeay() function returns a pointer to that method.

The RAND_set_rand_method() function sets the RAND method to meth. The RAND_get_rand_method() function returns a pointer to the current method.

RAND_method Structure

```
 typedef struct rand_meth_st
 {
        void (*seed)(const void *buf, int num);
        int (*bytes)(unsigned char *buf, int num);
        void (*cleanup)(void);
        void (*add)(const void *buf, int num, int entropy);
        int (*pseudorand)(unsigned char *buf, int num);

int (*status)(void);
 } RAND_METHOD;
```

The components point to the implementation of the RAND_seed(), RAND_bytes(), RAND_cleanup(), RAND_add(), RAND_pseudo_rand(), and RAND_status() functions. Each component may be NULL if the function is not implemented.

RETURN VALUES

The RAND_set_rand_method() function returns no value. The RAND_get_rand_method() and RAND_SSLeay() functions return pointers to the respective methods.

HISTORY

The `RAND_set_rand_method()`, `RAND_get_rand_method()`, and `RAND_SSLeay()` functions are available in all versions of OpenSSL.

SEE ALSO

Functions: *rand*

rand_ssl

NAME

rand_ssl – Pseudo-random number generator

SYNOPSIS

```
#include <openssl/rand.h>
int RAND_bytes(
        unsigned char *buf, int num
);
int RAND_pseudo_bytes(
        unsigned char *buf, int num
);
void RAND_seed(
        const void *buf, int num
);
void RAND_add(
        const void *buf, int num, int entropy
);
int RAND_status(
        void
);
void RAND_screen(
        void
);
int RAND_load_file(
        const char *file, long max_bytes
);
int RAND_write_file(
        const char *file
);
const char *RAND_file_name(
         char *file, size_t num
);
int RAND_egd(
        const char *path
);
void RAND_set_rand_method(
         RAND_METHOD *meth
);
```

```
RAND_METHOD *RAND_get_rand_method(
        void
);
RAND_METHOD *RAND_SSLeay(
         void
);
void RAND_cleanup(
        void
);
```

DESCRIPTION

These functions implement a cryptographically secure pseudo-random number generator (PRNG). It is used by other library functions, for example, to generate random keys. Applications can use it when they need randomness.

A cryptographic PRNG must be seeded with unpredictable data, such as mouse movements or keys pressed at random by the user. This is described in *RAND_add*. Its state can be saved in a seed file (see *RAND_load_file*) to avoid having to go through the seeding process whenever the application is started.

For more information on how to obtain random data from the PRNG, see *RAND_bytes*.

Internals

The `RAND_SSLeay()` method implements a PRNG based on a cryptographic hash function.

The following description of its design is based on the SSLeay documentation. A good RNG includes the following components:

1. A good hashing algorithm to mix things up and to convert the RNG state to random numbers.
2. An initial source of random state.
3. The state should be very large. If the RNG is used to generate 4096 bit RSA keys, two 2048-bit random strings are required (at a minimum). If your RNG state only has 128 bits, you are limiting the search space to 128 bits, not 2048. It should be easier to break a cipher than guess the RNG seed data.
4. Any RNG seed data should influence all subsequent random numbers generated. This implies that any random seed data entered will have an influence on all subsequent random numbers generated.
5. When using data to seed the RNG state, the data should not be extractable from the RNG state. We believe this should be a requirement because one possible source of secret semi-random data would be a private key or a password. This data must not be disclosed by either subsequent random numbers or a core dump left by a program crash.
6. Given the same initial state, two systems should deviate in their RNG state (and hence the random numbers generated) over time if at all possible.
7. Given the random number output stream, it should not be possible to determine the RNG state or the next random number.

The algorithm is as follows.

There is global state made up of a 1023 byte buffer (the state), a working hash value (md), and a counter (count).

Whenever seed data is added, it is inserted into the state as follows:

The input is divided into blocks of 20 bytes (or less for the last block). Each block is run through the hash function as follows: The data passed to the hash function is the current md, the same number of bytes from the state (the location determined by an incremented looping index) as the current block, the new key data block, and count (which is incremented after each use). The result of this is kept in md and also xored into the state at the same locations that were used as input into the hash function. This system addresses points 1 (hash function; currently SHA-1), 3 (the state), 4 (via the md), and 5 (by the use of a hash function and xor).

When bytes are extracted from the RNG, the following process is used. For each group of 10 bytes (or less), you do the following:

Input into the hash function the local md (which is initialized from the global md before any bytes are generated), the bytes that are to be overwritten by the random bytes, and bytes from the state (incrementing looping index). From this digest output (which is kept in md), the top (up to) 10 bytes are returned to the caller and the bottom 10 bytes are xored into the state.

Finally, after you finish num random bytes for the caller, count (which is incremented) and the local and global md are fed into the hash function and the results are kept in the global md.

I believe the above addressed points 1 (use of SHA-1), 6 (by hashing into the 'state' the 'old' data from the caller that is about to be overwritten) and 7 (by not using the 10 bytes given to the caller to update the 'state', but they are used to update 'md').

Of the points raised, only the second is not addressed (see *RAND_add*).

SEE ALSO

Functions: *BN_rand*, *RAND_add*, *RAND_load_file*, *RAND_egd*, *RAND_bytes*, *RAND_set_rand_method*, *RAND_cleanup, rand*

rc4

NAME

rc4: RC4_set_key, RC4 – RC4 encryption

SYNOPSIS

```
#include <openssl/rc4.h>

void RC4_set_key(

        RC4_KEY *key, int len, const unsigned char *data

);

void RC4(

        RC4_KEY *key, unsigned long len, const unsigned char *indata, unsigned char
        *outdata

);
```

DESCRIPTION

This library implements the Alleged RC4 cipher, which is described in *Applied Cryptography*. It is believed to be compatible with RC4™, a proprietary cipher of RSA Security Inc.

RC4 is a stream cipher with variable key length. Typically, 128 bit (16 byte) keys are used for strong encryption, but shorter insecure key sizes have been widely used due to export restrictions.

RC4 consists of a key setup phase and the actual encryption or decryption phase.

The `RC4_set_key()` function sets up the `RC4_KEY` key using the `len` bytes long key at `data`.

The `RC4()` function encrypts or decrypts the `len` bytes of data at `indata` using `key` and places the result at `outdata`. Repeated `RC4()` calls with the same `key` yield a continuous key stream.

Since RC4 is a stream cipher (the input is XORed with a pseudo-random key stream to produce the output), decryption uses the same function calls as encryption.

Applications should use the higher level functions instead of calling the RC4 functions directly. (See *EVP_EncryptInit*.)

NOTES

Certain conditions have to be observed to securely use stream ciphers. You cannot perform multiple encryptions using the same key stream.

RETURN VALUES

The `RC4_set_key()` and `RC4()` functions do not return values.

HISTORY

The `RC4_set_key()` and `RC4()` functions are available in all versions of SSLeay and OpenSSL.

SEE ALSO

Functions: *blowfish*, *des*, *rc2*

req

NAME

req – PKCS#10 certificate and certificate generating utility

SYNOPSIS

openssl req [-inform PEM|DER] [-outform PEM|DER] [-in *filename*] [-passin *arg*] [-out *filename*] [-passout *arg*] [-text] [-noout] [-verify] [-modulus] [-new] [-rand *filename*] [-newkey rsa:bits] [-newkey dsa:file] [-nodes] [-key *filename*] [-keyform PEM|DER] [-keyout *filename*] [-md5 | sha1 | md2 | mdc2] [-config *filename*] [-x509] [-days *n*] [-asn1kludge] [-newhdr] [-extensions *section*] [-reqexts *section*]

OPTIONS

inform DER|PEM

Specifies the input format. The DER option uses an ASN1 DER encoded form compatible with the PKCS#10. The PEM form is the default format; it consists of the DER format base64 encoded with additional header and footer lines.

outform DER|PEM

Specifies the output format. The options have the same meaning as the inform option.

in *filename*

Specifies the input filename to read a request from or standard input if this option is not specified. A request is only read if the creation options (new and newkey) are not specified.

passin *arg*

Input file password source. For more information about the format of arg, see the Pass Phrase Arguments section in *openssl*.

out *filename*

Specifies the output filename to write to or standard output by default.

passout *arg*

Output file password source. For more information about the format of arg, see the Pass Phrase Arguments section in *openssl*.

text

Prints out the certificate request in text form.

noout

Prevents output of the encoded version of the request.

modulus

Prints out the value of the modulus of the public key contained in the request.

verify

Verifies the signature on the request.

new

Generates a new certificate request. It will prompt the user for the relevant field values. The actual fields prompted for and their maximum and minimum sizes are specified in the configuration file and any requested extensions.

If the `key` option is not used it will generate a new RSA private key using information specified in the configuration file.

`rand` *`filename`*

A file or files containing random data used to seed the random number generator, or an EGD socket. (See *RAND_egd*.) Multiple files are separated by an OS-dependent character. The separator is a semicolon (;) for MS-Windows, a comma (,) for OpenVMS, and a colon (:) for all others.

`newkey` *`arg`*

Creates a new certificate request and a new private key. The argument takes one of two forms. The `rsa:nbits`, where `nbits` is the number of bits, generates an RSA key `nbits` in size. The `dsa:filename` generates a DSA key using the parameters in the file *`filename`*.

`key` *`filename`*

Specifies the file to read the private key from. It also accepts PKCS#8 format private keys for `PEM` format files.

`keyform PEM|DER`

The format of the private key file specified in the `key` option. The `PEM` format is the default.

`keyout` *`filename`*

Gives the filename to write the newly created private key to. If this option is not specified then the filename present in the configuration file is used.

`nodes`

If this option is specified then if a private key is created it will not be encrypted.

`[md5|sha1|md2| mdc2]`

Specifies the message digest to sign the request with. This overrides the digest algorithm specified in the configuration file. This option is ignored for DSA requests; they always use SHA1.

`config` *`filename`*

Allows an alternative configuration file to be specified. This overrides the compile time filename or any specified in the `OPENSSL_CONF` environment variable.

`x509`

Outputs a self-signed certificate instead of a certificate request. This is typically used to generate a test certificate or a self-signed root CA. The extensions added to the certificate (if any) are specified in the configuration file.

`days` *`n`*

When the `x509` option is being used this specifies the number of days to certify the certificate. The default is 30 days.

`extensions` *`section`*

`reqexts` *`section`*

Specifies alternative sections to include certificate extensions (if the `x509` option is present) or certificate request extensions. This allows several different sections to be used in the same configuration file to specify requests for a variety of purposes.

`asn1kludge`

By default the `req` command outputs certificate requests containing no attributes in the correct PKCS#10 format. However certain CAs will only accept requests containing no attributes in an invalid form. This option produces this invalid format.

More precisely the `Attributes` in a PKCS#10 certificate request are defined as a `SET OF Attribute`. They are not optional, so if no attributes are present then they should be encoded as an empty `SET OF`. The invalid form does not include the empty `SET OF`, whereas the correct form does.

It should be noted that very few CAs still require the use of this option.

`newhdr`

Adds the word `NEW` to the `PEM` file header and footer lines on the outputed request. Some software (Netscape certificate server) and some CAs need this.

CONFIGURATION OPTIONS

The configuration options are specified in the `req` section of the configuration file. As with all configuration files if no value is specified in the specific section (i.e. `req`) then the initial unnamed or `default` section is searched too.

The options available are described in detail below.

`input_password output_password`

The passwords for the input private key file (if present) and the output private key file (if one will be created). The command line options `passin` and `passout` override the configuration file values.

`default_bits`

Specifies the default key size in bits. If not specified then 512 is used. It is used if the `new` option is used. It can be overridden by using the `newkey` option.

`default_keyfile`

The default filename to write a private key to. If not specified the key is written to standard output. This can be overridden by the `keyout` option.

`oid_file`

Specifies a file containing additional object identifiers. Each line of the file should consist of the numerical form of the object identifier followed by white space then the short name followed by white space and finally the long name.

`oid_section`

Specifies a section in the configuration file containing extra object identifiers. Each line should consist of the short name of the object identifier followed by = and the numerical form. The short and long names are the same when this option is used.

`RANDFILE`

Specifies a filename in which random number seed information is placed and read from, or an EGD socket. (See *RAND_egd*). It is used for private key generation.

`encrypt_key`

If this is set to `no` then if a private key is generated it is not encrypted. This is equivalent to the `nodes` command line option. For compatibility `encrypt_rsa_key` is an equivalent option.

`default_md`

Specifies the digest algorithm to use. Possible values include `md5 sha1 mdc2`. If not present then `MD5` is used. This option can be overridden on the command line.

`string_mask`

Masks out the use of certain string types in certain fields. Most users will not need to change this option.

It can be set to several values. The `default` option uses PrintableStrings, T61Strings and BMPStrings. If the `pkix` value is used then only PrintableStrings and BMPStrings will be used. This follows the PKIX recommendation in RFC2459. If the `utf8only` option is used then only UTF8Strings will be used; this is the PKIX recommendation in RFC2459 after 2003. Finally, the `nombstr` option uses PrintableStrings and T61Strings. Certain software has problems with BMPStrings and UTF8Strings, particularly Netscape.

`req_extensions`

Specifies the configuration file section containing a list of extensions to add to the certificate request. It can be overridden by the `reqexts` command line option.

`x509_extensions`

Specifies the configuration file section containing a list of extensions to add to the certificate generated when the `x509` option is used. It can be overridden by the `extensions` command line option.

`prompt`

If set to the value `no` this disables prompting of certificate fields and takes values from the config file directly. It also changes the expected format of the `distinguished_name` and `attributes` sections.

`attributes`

Specifies the section containing any request attributes. Its format is the same as `distinguished_name`. Typically these may contain the `challengePassword` or `unstructuredName` types. They are currently ignored by OpenSSL's request signing utilities but some CAs might want them.

`distinguished_name`

Specifies the section containing the distinguished name fields to prompt for when generating a certificate or certificate request. The format is described in the next section.

DISTINGUISHED NAME AND ATTRIBUTES SECTIONS FORMAT

There are two separate formats for the distinguished name and attribute sections. If the `prompt` option is set to `no` then these sections only consist of field names and values. An example follows:

```
CN=My Name
OU=My Organization
emailAddress=someone@somewhere.org
```

This allows external programs (e.g. GUI based) to generate a template file with all the field names and values and pass it to `req`. An example of this kind of configuration file is contained in the Examples section.

Alternatively if the `prompt` option is absent or not set to `no` then the file contains field prompting information. It consists of lines such as the following:

```
fieldName="prompt"
fieldName_default="default field value"
fieldName_min= 2
fieldName_max= 4
```

The fieldName is the field name being used, such as commonName (or CN).

The prompt string is used to ask the user to enter the relevant details. If the user enters nothing then the default value is used. If no default value is present then the field is omitted. A field can still be omitted if a default value is present if the user enters the '.' character.

The number of characters entered must be between the fieldName_min and fieldName_max limits. There may be additional restrictions based on the field being used. For example, countryName can only be two characters long and must fit in a PrintableString.

Some fields, such as organizationName, can be used more than once in a DN. This presents a problem because configuration files will not recognize the same name occurring twice. To avoid this problem if the fieldName contains some characters followed by a full stop they will be ignored. So, for example, a second organizationName can be input by calling it `1.organizationName`.

The actual permitted field names are any object identifier short or long names. These are compiled into OpenSSL and include the usual values such as commonName, countryName, localityName, organizationName, organizationUnitName, stateOrPrivinceName. Additionally emailAddress is included, as well as name, surname, givenName initials and dnQualifier.

Additional object identifiers can be defined with the `oid_file` or `oid_section` options in the configuration file. Any additional fields will be treated as though they were a DirectoryString.

DESCRIPTION

The `req` command primarily creates and processes certificate requests in PKCS#10 format. It can additionally create self signed certificates for use as root CAs for example.

NOTES

The header and footer lines in the `PEM` format are normally:

```
-----BEGIN CERTIFICATE REQUEST----
-----END CERTIFICATE REQUEST----
```

Some software, including some versions of Netscape certificate server, need:

```
-----BEGIN NEW CERTIFICATE REQUEST----
-----END NEW CERTIFICATE REQUEST----
```

This is produced with the `newhdr` option, but is otherwise compatible. Either form is accepted transparently on input.

The certificate requests generated by `Xenroll` with MSIE have extensions added. It includes the `keyUsage` extension which determines the type of key (signature only or general purpose) and any additional OIDs entered by the script in an extendedKeyUsage extension.

RESTRICTIONS

OpenSSL's handling of T61Strings (also known as TeletexStrings) is broken. It effectively treats them as ISO-8859-1 (Latin 1). Netscape and MSIE have similar behavior. This can cause problems if you need characters that are not available in PrintableStrings and you do not want to or cannot use BMPStrings.

As a consequence of the T61String handling the only correct way to represent accented characters in OpenSSL is to use a BMPString. Unfortunately Netscape currently chokes on these. If you have to use accented characters with Netscape and MSIE then you currently need to use the invalid T61String form.

The current prompting is not very friendly. It doesn't allow you to confirm what you've entered. Other things, such as extensions in certificate requests, are statically defined in the configuration file. Some of these, such as an email address in subjectAltName, should be input by the user.

ERRORS

The following messages are frequently asked about:

```
Using configuration from /some/path/openssl.cnf
Unable to load config info
```

This is followed some time later by the following lines:

```
unable to find 'distinguished_name' in config
problems making Certificate Request
```

The first error message is the clue. It means the configuration file cannot be found. Certain operations, such as examining a certificate request, do not need a configuration file; so its use isn't enforced. Generation of certificates or requests, however, do need a configuration file.

Another error message is this:

```
        Attributes:
            a0:00
```

This is displayed when no attributes are present and the request includes the correct empty `SET OF` structure (the DER encoding of which is 0xa0 0x00). If you only see

```
        Attributes:
```

then the `SET OF` is missing and the encoding is technically invalid (but it is tolerated). See the description of the command line option `asn1kludge` for more information.

EXAMPLES

Examine and verify certificate request:

```
openssl req -in req.pem -text -verify -noout
```

Create a private key and then generate a certificate request from it:

```
openssl genrsa -out key.pem 1024
openssl req -new -key key.pem -out req.pem
```

The same but just using req:

```
openssl req -newkey rsa:1024 -keyout key.pem -out req.pem
```

Generate a self signed root certificate:

```
openssl req -x509 -newkey rsa:1024 -keyout key.pem -out req.pem
```

Example of a file pointed to by the `oid_file` option:

```
1.2.3.4 shortNameA longer Name
1.2.3.6 otherNameOther longer Name
```

Example of a section pointed to by `oid_section` making use of variable expansion:

```
testoid1=1.2.3.5
testoid2=${testoid1}.6
```

Sample configuration file prompting for field values:

```
[ req ]
 default_bits= 1024
 default_keyfile = privkey.pem
 distinguished_name= req_distinguished_name
 attributes= req_attributes
 x509_extensions= v3_ca

 dirstring_type = nobmp

 [ req_distinguished_name ]
 countryName= Country Name (2 letter code)
 countryName_default= AU
 countryName_min= 2
 countryName_max= 2

 localityName= Locality Name (eg, city)

 organizationalUnitName= Organizational Unit Name (eg, section)
 commonName= Common Name (eg, YOUR name)
 commonName_max= 64
 emailAddress= Email Address
 emailAddress_max= 40

 [ req_attributes ]

 challengePassword= A challenge password
 challengePassword_min= 4
 challengePassword_max= 20

 [ v3_ca ]

 subjectKeyIdentifier=hash
 authorityKeyIdentifier=keyid:always,issuer:always
 basicConstraints = CA:true
```

Sample configuration containing all field values:

```
 RANDFILE= $ENV::HOME/.rnd

 [ req ]

 default_bits= 1024
 default_keyfile = keyfile.pem
 distinguished_name= req_distinguished_name
 attributes= req_attributes
 prompt= no
 output_password= mypass

 [ req_distinguished_name ]
```

```
C= GB
ST= Test State or Province
L= Test Locality
O= Organization Name
OU= Organizational Unit Name
CN= Common Name
emailAddress= test@email.address

[ req_attributes ]

challengePassword= A challenge password
```

ENVIRONMENT VARIABLES

If defined, the variable `OPENSSL_CONF` allows an alternative configuration file location to be specified. It will be overridden by the `config` command line option if it is present. For compatibility reasons the `SSLEAY_CONF` environment variable serves the same purpose, but its use is discouraged.

SEE ALSO

Commands: *x509*, *ca*, *genrsa*, *gendsa*

Others: *config*

RIPEMD160

NAME

RIPEMD160, RIPEMD160_Init, RIPEMD160_Update, RIPEMD160_Final – RIPEMD-160 hash function

SYNOPSIS

```
#include <openssl/ripemd.h>
unsigned char *RIPEMD160(
        const unsigned char *d, unsigned long n, unsigned char *md
);
void RIPEMD160_Init(
       RIPEMD160_CTX *c
);
void RIPEMD160_Update(
       RIPEMD_CTX *c, const void *data, unsigned long len
);
void RIPEMD160_Final(
       unsigned char *md, RIPEMD160_CTX *c
);
```

DESCRIPTION

RIPEMD-160 is a cryptographic hash function with a 160 bit output.

The RIPEMD160() function computes the RIPEMD-160 message digest of the n bytes at d and places it in md (which must have space for RIPEMD160_DIGEST_LENGTH == 20 bytes of output). If md is NULL, the digest is placed in a static array.

The following functions may be used if the message is not completely stored in memory:

The RIPEMD160_Init() function initializes a RIPEMD160_CTX structure.

The RIPEMD160_Update() can be called repeatedly with chunks of the message to be hashed (len bytes at data).

The RIPEMD160_Final() function places the message digest in md, which must have space for RIPEMD160_DIGEST_LENGTH == 20 bytes of output, and erases the RIPEMD160_CTX.

Applications should use the higher level functions , such as EVP_DigestInit(), instead of calling the hash functions directly.

The ripemd() function conforms to ISO/IEC 10118-3 (draft).

RETURN VALUES

The RIPEMD160() function returns a pointer to the hash value.

The RIPEMD160_Init(), RIPEMD160_Update(), and RIPEMD160_Final() functions do not return values.

HISTORY

The `RIPEMD160()`, `RIPEMD160_Init()`, `RIPEMD160_Update()`, and `RIPEMD160_Final()` functions are available since SSLeay 0.9.0.

SEE ALSO

Functions: *sha*, *hmac*, *EVP_DigestInit*

rsa

NAME

rsa – RSA public key cryptosystem

SYNOPSIS

```
#include <openssl/rsa.h>
#include <openssl/engine.h>

RSA * RSA_new(
       void
);

void RSA_free(
       RSA *rsa
);

int RSA_public_encrypt(
       int flen, unsigned char *from, unsigned char *to, RSA *rsa, int padding
);

int RSA_private_decrypt(
        int flen, unsigned char *from, unsigned char *to, RSA *rsa, int padding
);

int RSA_sign(
       int type, unsigned char *m, unsigned int m_len, unsigned char *sigret, unsigned
       int *siglen, RSA *rsa
);

int RSA_verify(
       int type, unsigned char *m, unsigned int m_len, unsigned char *sigbuf, unsigned
       int siglen, RSA *rsa
);

RSA *RSA_generate_key(
       int num, unsigned long e, void (*callback)(int,int,void *), void *cb_arg
);

int RSA_check_key(
       RSA *rsa
);

int RSA_blinding_on(
       RSA *rsa, BN_CTX *ctx
);

void RSA_blinding_off(
       RSA *rsa
);

void RSA_set_default_openssl_method(
```

```
        RSA_METHOD *meth
);
RSA_METHOD *RSA_get_default_openssl_method(
        void
);
int RSA_set_method(
        RSA *rsa, ENGINE *engine
);
RSA_METHOD *RSA_get_method(
         RSA *rsa
);
RSA_METHOD *RSA_PKCS1_SSLeay(
         void
);
RSA_METHOD *RSA_PKCS1_RSAref(
         void
);
RSA_METHOD *RSA_null_method(
         void
);
int RSA_flags(
        RSA *rsa
);
RSA *RSA_new_method(
        ENGINE *engine
);
int RSA_print(
        BIO *bp, RSA *x, int offset
);
int RSA_print_fp(
        FILE *fp, RSA *x, int offset
);
int RSA_get_ex_new_index(
         long argl, char *argp, int (*new_func)(), int (*dup_func)(), void (*free_func)()
);
int RSA_set_ex_data(
        RSA *r,int idx,char *arg
);
char *RSA_get_ex_data(
        RSA *r, int idx
```

```
);
int RSA_private_encrypt(
        int flen, unsigned char *from, unsigned char *to, RSA *rsa,int padding
);
int RSA_public_decrypt(
        int flen, unsigned char *from, unsigned char *to, RSA *rsa,int padding
);
int RSA_sign_ASN1_OCTET_STRING(
        int dummy, unsigned char *m, unsigned int m_len, unsigned char *sigret, unsigned
        int *siglen, RSA *rsa
);
int RSA_verify_ASN1_OCTET_STRING(
        int dummy, unsigned char *m, unsigned int m_len, unsigned char *sigbuf, unsigned
        int siglen, RSA *rsa
);
```

DESCRIPTION

These functions implement RSA public key encryption and signatures as defined in PKCS #1 v2.0 [RFC 2437].

The RSA structure consists of several BIGNUM components. It can contain public as well as private RSA keys:

```
 struct
        {
        BIGNUM *n;// public modulus
        BIGNUM *e;// public exponent
        BIGNUM *d;// private exponent
        BIGNUM *p;// secret prime factor
        BIGNUM *q;// secret prime factor
        BIGNUM *dmp1;// d mod (p-1)
        BIGNUM *dmq1;// d mod (q-1)
        BIGNUM *iqmp;// q^-1 mod p
// ...
        };

 RSA
```

In public keys, the private exponent and the related secret values are NULL.

The p, q, dmp1, dmq1 and iqmp may be NULL in private keys, but the RSA operations are much faster when these values are available.

The rsa() function conforms to SSL, PKCS #1 v2.0. It was covered by a US patent which expired in September 2000.

SEE ALSO

Commands: *rsa*

Functions: *bn*, *dsa*, *dh*, *rand*, *RSA_new*, *RSA_public_encrypt*, *RSA_sign*, *RSA_size*, *RSA_generate_key*, *RSA_check_key*, *RSA_blinding_on*, *RSA_set_method*, *RSA_print*, *RSA_get_ex_new_index*, *RSA_private_encrypt*, *RSA_sign_ASN_OCTET_STRING*, *RSA_padding_add_PKCS1_type_1*

RSA_blinding_on

NAME

RSA_blinding_on, RSA_blinding_off – Protect the RSA operation from timing attacks

SYNOPSIS

```
#include <openssl/rsa.h>
int RSA_blinding_on(
       RSA *rsa, BN_CTX *ctx
);
void RSA_blinding_off(
       RSA *rsa
);
```

DESCRIPTION

RSA is vulnerable to timing attacks. In a setup where attackers can measure the time of RSA decryption or signature operations, blinding must be used to protect the RSA operation from that attack.

The `RSA_blinding_on()` function turns blinding on for key `rsa` and generates a random blinding factor. The `ctx` is `NULL` or a pre-allocated and initialized `BN_CTX`. The random number generator must be seeded prior to calling the `RSA_blinding_on()` function.

The `RSA_blinding_off()` function turns blinding off and frees the memory used for the blinding factor.

RETURN VALUES

The `RSA_blinding_on()` function returns 1 on success, and 0 if an error occurred.

The `RSA_blinding_off()` function returns no value.

HISTORY

The `RSA_blinding_on()` and `RSA_blinding_off()` functions appeared in SSLeay 0.9.0.

SEE ALSO

Functions: *rsa*, *rand*

RSA_check_key

NAME

RSA_check_key – Validate private RSA keys

SYNOPSIS

```
#include <openssl/rsa.h>
int RSA_check_key(
       RSA *rsa
);
```

DESCRIPTION

This function validates RSA keys. It checks that `p` and `q` are prime, and that `n = p*q`.

It also checks that `d*e = 1 mod (p-1*q-1)`, and that `dmp1`, `dmq1` and `iqmp` are set correctly or are `NULL`.

The key's public components may not be `NULL`.

RETURN VALUE

The `RSA_check_key()` function returns 1 if `rsa` is a valid RSA key, and 0 otherwise. If an error occurs while checking the key -1 is returned.

If the key is invalid or an error occurred, the reason code can be obtained using the `ERR_get_error()` function.

HISTORY

The `RSA_check()` function appeared in OpenSSL 0.9.4.

SEE ALSO

Functions: *rsa*, *err*

RSA_generate_key

NAME

RSA_generate_key – Generate RSA key pair

SYNOPSIS

```
#include <openssl/rsa.h>
RSA *RSA_generate_key(
       int num, unsigned long e, void (*callback)(int,int,void *), void *cb_arg
);
```

DESCRIPTION

The `RSA_generate_key()` function generates a key pair and returns it in a newly allocated `RSA` structure. The pseudo-random number generator must be seeded prior to calling `RSA_generate_key()`.

The modulus size will be `num` bits, and the public exponent will be `e`. Key sizes with `num` < 1024 should be considered insecure. The exponent is an odd number, typically 3 or 65535.

A callback function may be used to provide feedback about the progress of the key generation. If `callback` is not `NULL`, it will be called as follows:

- While a random prime number is generated, it is called as described in *BN_generate_prime*.
- When the n-th randomly generated prime is rejected as not suitable for the key, `callback(2, n, cb_arg)` is called.
- When a random p has been found with p-1 relatively prime to `e`, it is called as `callback(3, 0, cb_arg)`.

The process is then repeated for prime q with `callback(3, 1, cb_arg)`.

RESTRICTIONS

`callback(2, x, cb_arg)` is used with two different meanings.

The `RSA_generate_key()` function goes into an infinite loop for illegal input values.

RETURN VALUE

If key generation fails, `RSA_generate_key()` returns `NULL`; the error codes can be obtained by using the `ERR_get_error()` function.

HISTORY

The `cb_arg` argument was added in SSLeay 0.9.0.

SEE ALSO

Functions: *err*, *rand*, *rsa*, *RSA_free*

RSA_get_ex_new_index

NAME

RSA_get_ex_new_index, RSA_set_ex_data, RSA_get_ex_data – Add application specific data to RSA structures

SYNOPSIS

```
#include <openssl/rsa.h>

int RSA_get_ex_new_index(

         long argl, void *argp, CRYPTO_EX_new *new_func, CRYPTO_EX_dup *dup_func,
        CRYPTO_EX_free *free_func

);

int RSA_set_ex_data(

        RSA *r, int idx, void *arg

);

void *RSA_get_ex_data(

        RSA *r, int idx

);

typedef int new_func(

        void *parent, void *ptr, CRYPTO_EX_DATA *ad, int idx, long argl, void *argp

);

typedef void free_func(

        void *parent, void *ptr, CRYPTO_EX_DATA *ad, int idx, long argl, void *argp

);

typedef int dup_func(

        CRYPTO_EX_DATA *to, CRYPTO_EX_DATA *from, void *from_d, int idx, long argl, void
        *argp

);
```

DESCRIPTION

Several OpenSSL structures can have application specific data attached to them. This has several potential uses, it can be used to cache data associated with a structure (for example the hash of some part of the structure) or some additional data (for example a handle to the data in an external library).

Since the application data can be anything at all, it is passed and retrieved as a `void *` type.

The `RSA_get_ex_new_index()` function is initially called to register some new application specific data. It takes three optional function pointers which are called when the parent structure (in this case an RSA structure) is initially created, when it is copied and when it is freed up. If any or all of these function pointer arguments are not used they should be set to NULL. The `RSA_get_ex_new_index()` function also takes additional long and pointer parameters which will be passed to the supplied functions but which otherwise have no special meaning. It returns an `index` which should be stored (typically in a static variable) and passed used in the `idx` parameter in the remaining functions. Each successful call to `RSA_get_ex_new_index()` will return an index greater than any previously returned. This is important because the optional functions are called in order of increasing index value.

The RSA_set_ex_data() function is used to set application specific data. The data is supplied in the arg parameter and its precise meaning is up to the application.

The RSA_get_ex_data() function is used to retrieve application specific data. The data is returned to the application. This will be the same value as supplied to a previous RSA_set_ex_data() call.

The new_func() function is called when a structure is initially allocated, such as with the RSA_new() function. The parent structure members will not have any meaningful values at this point. This function will typically be used to allocate any application specific structure.

The free_func() function is called when a structure is being freed up. The dynamic parent structure members should not be accessed because they will be freed up when this function is called.

The new_func() and free_func() functions take the same parameters. The parent is a pointer to the parent RSA structure. The ptr is the application specific data, which is not very useful in new_func(). The ad is a pointer to the CRYPTO_EX_DATA structure from the parent RSA structure. The functions CRYPTO_get_ex_data() and CRYPTO_set_ex_data() can be called to manipulate it. The idx parameter is the index. This will be the same value returned by the RSA_get_ex_new_index() function when the functions were initially registered. Finally, the argl and argp parameters are the values originally passed to the same corresponding parameters when the RSA_get_ex_new_index() function was called.

The dup_func() function is called when a structure is being copied. Pointers to the destination and source CRYPTO_EX_DATA structures are passed in the to and from parameters respectively. The from_d parameter is passed a pointer to the source application data when the function is called. When the function returns, the value is copied to the destination. The application can thus modify the data pointed to by from_d and have different values in the source and destination. The idx, argl and argp parameters are the same as those in the new_func() and free_func() functions.

RESTRICTIONS

The dup_func() function is never called.

The return value of the new_func() function is ignored.

The new_func() function is not very useful because no meaningful values are present in the parent RSA structure when it is called.

RETURN VALUES

The RSA_get_ex_new_index() function returns a new index or -1 on failure (0 is a valid index value).

The RSA_set_ex_data() function returns 1 on success or 0 on failure.

The RSA_get_ex_data() function returns the application data or 0 on failure. 0 may also be valid application data but currently it can only fail if given an invalid idx parameter.

The new_func() and dup_func() functions should return 0 for failure and 1 for success.

On failure an error code can be obtained by using the ERR_get_error() function.

HISTORY

The RSA_get_ex_new_index(), RSA_set_ex_data(), and RSA_get_ex_data() functions are available since SSLeay 0.9.0.

SEE ALSO

Functions: *rsa*, *CRYPTO_set_ex_data*

RSA_new

NAME

RSA_new, RSA_free – Allocate and free RSA objects

SYNOPSIS

```
#include <openssl/rsa.h>
RSA * RSA_new(
        void
);
void RSA_free(
        RSA *rsa
);
```

DESCRIPTION

The `RSA_new()` function allocates and initializes an `RSA` structure.

The `RSA_free()` function frees the `RSA` structure and its components. The key is erased before the memory is returned to the system.

RETURN VALUES

If the allocation fails, the `RSA_new()` function returns `NULL` and sets an error code that can be obtained by using the `ERR_get_error()` function. Otherwise it returns a pointer to the newly allocated structure.

The `RSA_free()` function returns no value.

HISTORY

The `RSA_new()` and `RSA_free()` functions are available in all versions of SSLeay and OpenSSL.

SEE ALSO

Functions: *err*, *rsa*, *RSA_generate_key*

RSA_padding_add_PKCS1_type_1

NAME

RSA_padding_add_PKCS1_type_1, RSA_padding_check_PKCS1_type_1, RSA_padding_add_PKCS1_type_2, RSA_padding_check_PKCS1_type_2, RSA_padding_add_PKCS1_OAEP, RSA_padding_check_PKCS1_OAEP, RSA_padding_add_SSLv23, RSA_padding_check_SSLv23, RSA_padding_add_none, RSA_padding_check_none – Asymmetric encryption padding

SYNOPSIS

```
#include <openssl/rsa.h>

int RSA_padding_add_PKCS1_type_1(
        unsigned char *to, int tlen, unsigned char *f, int fl
);

int RSA_padding_check_PKCS1_type_1(
       unsigned char *to, int tlen, unsigned char *f, int fl, int rsa_len
);

int RSA_padding_add_PKCS1_type_2(
        unsigned char *to, int tlen, unsigned char *f, int fl
);

int RSA_padding_check_PKCS1_type_2(
       unsigned char *to, int tlen, unsigned char *f, int fl, int rsa_len
);

int RSA_padding_add_PKCS1_OAEP(
        unsigned char *to, int tlen, unsigned char *f, int fl, unsigned char *p, int pl
);

int RSA_padding_check_PKCS1_OAEP(
        unsigned char *to, int tlen, unsigned char *f, int fl, int rsa_len, unsigned
       char *p, int pl
);

int RSA_padding_add_SSLv23(
        unsigned char *to, int tlen, unsigned char *f, int fl
);

int RSA_padding_check_SSLv23(
        unsigned char *to, int tlen, unsigned char *f, int fl, int rsa_len
);

int RSA_padding_add_none(
        unsigned char *to, int tlen, unsigned char *f, int fl
);

int RSA_padding_check_none(
        unsigned char *to, int tlen, unsigned char *f, int fl, int rsa_len
);
```

DESCRIPTION

The RSA_padding_xxx_xxx() functions are called from the RSA encrypt, decrypt, sign and verify functions. Normally they should not be called from application programs.

However, they can also be called directly to implement padding for other asymmetric ciphers. The RSA_padding_add_PKCS1_OAEP() and RSA_padding_check_PKCS1_OAEP() function can be used in an application combined with RSA_NO_PADDING in order to implement OAEP with an encoding parameter.

The RSA_padding_add_xxx() functions encode fl bytes from f so as to fit into tlen bytes and stores the result at to. An error occurs if fl does not meet the size requirements of the encoding method.

The following encoding methods are implemented:

- PKCS1_type_1

  PKCS #1 v2.0 EMSA-PKCS1-v1_5 (PKCS #1 v1.5 block type 1); used for signatures

- PKCS1_type_2

  PKCS #1 v2.0 EME-PKCS1-v1_5 (PKCS #1 v1.5 block type 2)

- PKCS1_OAEP

  PKCS #1 v2.0 EME-OAEP

- SSLv23

  PKCS #1 EME-PKCS1-v1_5 with SSL-specific modification

- none

  simply copy the data

The random number generator must be seeded prior to calling the RSA_padding_add_xxx() functions.

The RSA_padding_check_xxx() functions verify that the fl bytes at f contain a valid encoding for an rsa_len byte RSA key in the respective encoding method. It then stores the recovered data of at most tlen bytes (for RSA_NO_PADDING: of size tlen) at to.

For RSA_padding_xxx_OAEP(), p points to the encoding parameter of length pl. The p may be NULL if pl is 0.

RETURN VALUES

The RSA_padding_add_xxx() functions return 1 on success, 0 on error. The RSA_padding_check_xxx() functions return the length of the recovered data, -1 on error. Error codes can be obtained by calling ERR_get_error().

HISTORY

The RSA_padding_add_PKCS1_type_1(), RSA_padding_check_PKCS1_type_1(), RSA_padding_add_PKCS1_type_2(), RSA_padding_check_PKCS1_type_2(), RSA_padding_add_SSLv23(), RSA_padding_check_SSLv23(), RSA_padding_add_none(), and RSA_padding_check_none() functions appeared in SSLeay 0.9.0.

The RSA_padding_add_PKCS1_OAEP() and RSA_padding_check_PKCS1_OAEP() functions were added in OpenSSL 0.9.2b.

RSA_print

NAME

RSA_print, RSA_print_fp, DSAparams_print, DSAparams_print_fp, DSA_printDSA_print_fp, DHparams_print, DHparams_print_fp – Print cryptographic parameters

SYNOPSIS

```
#include <openssl/rsa.h>
int RSA_print(
       BIO *bp, RSA *x, int offset
);
int RSA_print_fp(
       FILE *fp, RSA *x, int offset
);
#include <openssl/dsa.h>
int DSAparams_print(
       BIO *bp, DSA *x
);
int DSAparams_print_fp(
       FILE *fp, DSA *x
);
int DSA_print(
       BIO *bp, DSA *x, int offset
);
int DSA_print_fp(
       FILE *fp, DSA *x, int offset
);
#include <openssl/dh.h>
int DHparams_print(
       BIO *bp, DH *x
);
int DHparams_print_fp(
       FILE *fp, DH *x
);
```

DESCRIPTION

A human-readable hexadecimal output of the components of the RSA key, DSA parameters or key or DH parameters is printed to `bp` or `fp`.

The output lines are indented by `offset` spaces.

RETURN VALUES

These functions return 1 on success, 0 on error.

HISTORY

The RSA_print(), RSA_print_fp(), DSA_print(), DSA_print_fp(), DH_print(), and DH_print_fp() functions are available in all versions of SSLeay and OpenSSL. The DSAparams_print() and DSAparams_print_pf() functions were added in SSLeay 0.8.

SEE ALSO

Functions: *dh*, *dsa*, *rsa*, *BN_bn2bin*

RSA_private_encrypt

NAME

RSA_private_encrypt, RSA_public_decrypt – Low level signature operations

SYNOPSIS

```
#include <openssl/rsa.h>
int RSA_private_encrypt(
        int flen, unsigned char *from, unsigned char *to, RSA *rsa, int padding
);
int RSA_public_decrypt(
       int flen, unsigned char *from, unsigned char *to, RSA *rsa, int padding
);
```

DESCRIPTION

These functions handle RSA signatures at a low level.

The `RSA_private_encrypt()` function signs the `flen` bytes at `from` (usually a message digest with an algorithm identifier) using the private key `rsa` and stores the signature in `to`. The `to` must point to `RSA_size(rsa)` bytes of memory.

The `padding` denotes one of the following modes:

RSA_PKCS1_PADDING

PKCS #1 v1.5 padding. This function does not handle the `algorithmIdentifier` specified in PKCS #1. When generating or verifying PKCS #1 signatures, the `RSA_sign()` and `RSA_verify()` functions should be used.

RSA_NO_PADDING

Raw RSA signature. This mode should only be used to implement cryptographically sound padding modes in the application code. Signing user data directly with RSA is insecure.

The `RSA_public_decrypt()` function recovers the message digest from the `flen` bytes long signature at `from` using the signer's public key `rsa`. The `to` must point to a memory section large enough to hold the message digest (which is smaller than `RSA_size(rsa) - 11`). The `padding` is the padding mode that was used to sign the data.

RETURN VALUES

The `RSA_private_encrypt()` function returns the size of the signature (i.e., `RSA_size(rsa)`). The `RSA_public_decrypt()` function returns the size of the recovered message digest.

On error, -1 is returned; the error codes can be obtained by using the `ERR_get_error()` function.

HISTORY

The `padding` argument was added in SSLeay 0.8. RSA_NO_PADDING is available since SSLeay 0.9.0.

RSA_public_encrypt

NAME

RSA_public_encrypt, RSA_private_decrypt – RSA public key cryptography

SYNOPSIS

```
#include <openssl/rsa.h>
int RSA_public_encrypt(
       int flen, unsigned char *from, unsigned char *to, RSA *rsa, int padding
);
int RSA_private_decrypt(
       int flen, unsigned char *from, unsigned char *to, RSA *rsa, int padding
);
```

DESCRIPTION

The `RSA_public_encrypt()` function encrypts the `flen` bytes at `from` (usually a session key) using the public key `rsa` and stores the ciphertext in `to`. The `to` must point to `RSA_size(rsa)` bytes of memory.

The `padding` denotes one of the following modes:

RSA_PKCS1_ PADDING

PKCS #1 v1.5 padding. This currently is the most widely used mode.

RSA_PKCS1_OAEP_ PADDING

EME-OAEP as defined in PKCS #1 v2.0 with SHA-1, MGF1 and an empty encoding parameter. This mode is recommended for all new applications.

RSA_SSLV23_ PADDING

PKCS #1 v1.5 padding with an SSL-specific modification that denotes that the server is SSL3 capable.

RSA_NO_PADDING

Raw RSA encryption. This mode should only be used to implement cryptographically sound padding modes in the application code. Encrypting user data directly with RSA is insecure.

The `flen` must be less than `RSA_size(rsa) - 11` for the PKCS #1 v1.5 based padding modes, and less than `RSA_size(rsa) - 41` for RSA_PKCS1_OAEP_PADDING. The random number generator must be seeded prior to calling the `RSA_public_encrypt()` function.

The `RSA_private_decrypt()` function decrypts the `flen` bytes at `from` using the private key `rsa` and stores the plain text in `to`. The `to` must point to a memory section large enough to hold the decrypted data , which is smaller than `RSA_size(rsa)`. The `padding` is the padding mode that was used to encrypt the data.

These functions conform to SSL, PKCS #1 v2.0.

NOTES

The *RSA_PKCS1_RSAref* method supports only the RSA_PKCS1_PADDING mode.

RETURN VALUES

The `RSA_public_encrypt()` function returns the size of the encrypted data, `RSA_size(rsa)`. The `RSA_private_decrypt()` function returns the size of the recovered plaintext.

On error, -1 is returned; the error codes can be obtained by using the `ERR_get_error()` function.

HISTORY

The `padding` argument was added in SSLeay 0.8. RSA_NO_PADDING is available since SSLeay 0.9.0, OAEP was added in OpenSSL 0.9.2b.

SEE ALSO

Functions: *err*, *rand*, *rsa*, *RSA_size*

RSA_set_method

NAME

RSA_set_method, RSA_get_method, RSA_set_default_openssl_method, RSA_get_default_openssl_method, RSA_PKCS1_SSLeay, RSA_PKCS1_RSAref, RSA_null_method, RSA_flags, RSA_new_method – Select RSA method

SYNOPSIS

```
#include <openssl/rsa.h>
#include <openssl/engine.h>

void RSA_set_default_openssl_method(
       RSA_METHOD *meth
);

RSA_METHOD *RSA_get_default_openssl_method(
       void
);

RSA_METHOD *RSA_set_method(
        RSA *rsa, ENGINE *engine
);

RSA_METHOD *RSA_get_method(
        RSA *rsa
);

RSA_METHOD *RSA_PKCS1_SSLeay(
        void
);

RSA_METHOD *RSA_PKCS1_RSAref(
        void
);

RSA_METHOD *RSA_null_method(
        void
);

int RSA_flags(
       RSA *rsa
);

RSA *RSA_new_method(
       ENGINE *engine
);
```

DESCRIPTION

An `RSA_METHOD` specifies the functions that OpenSSL uses for RSA operations. By modifying the method, alternative implementations such as hardware accelerators can be used.

Initially, the default is to use the OpenSSL internal implementation, unless OpenSSL was configured with the `rsaref` or `-DRSA_NULL` options. The `RSA_PKCS1_SSLeay()` function returns a pointer to that method.

The `RSA_PKCS1_RSAref()` function returns a pointer to a method that uses the RSAref library. This is the default method in the `rsaref` configuration; the function is not available in other configurations. The `RSA_null_method()` function returns a pointer to a method that does not support the RSA transformation. It is the default if OpenSSL is compiled with `-DRSA_NULL`. These methods can be useful in the USA because of a patent on the RSA cryptosystem.

The `RSA_set_default_openssl_method()` function makes `meth` the default method for all `RSA` structures created later. However, this is true only when the default engine for RSA operations remains as openssl. ENGINEs provide an encapsulation for implementations of one or more algorithms at a time, and all the RSA functions mentioned here operate within the scope of the default openssl engine.

The `RSA_get_default_openssl_method()` function returns a pointer to the current default method for the openssl engine.

The `RSA_set_method()` function selects `engine` for all operations using the key `rsa`.

The `RSA_get_method()` function returns a pointer to the RSA_METHOD from the currently selected ENGINE for `rsa`.

The `RSA_flags()` function returns the `flags` that are set for `rsa`'s current method.

The `RSA_new_method()` function allocates and initializes an RSA structure so that `engine` will be used for the RSA operations. If `engine` is NULL, the default engine for RSA operations is used.

RSA_METHOD Structure

```
 typedef struct rsa_meth_st
 {
     /* name of the implementation */

const char *name;

     /* encrypt */

int (*rsa_pub_enc)(int flen, unsigned char *from,
          unsigned char *to, RSA *rsa, int padding);

     /* verify arbitrary data */

int (*rsa_pub_dec)(int flen, unsigned char *from,
          unsigned char *to, RSA *rsa, int padding);

     /* sign arbitrary data */

int (*rsa_priv_enc)(int flen, unsigned char *from,
          unsigned char *to, RSA *rsa, int padding);

     /* decrypt */

int (*rsa_priv_dec)(int flen, unsigned char *from,
          unsigned char *to, RSA *rsa, int padding);

     /* compute r0 = r0 ^ I mod rsa->n (May be NULL for some
                                        implementations) */

int (*rsa_mod_exp)(BIGNUM *r0, BIGNUM *I, RSA *rsa);
```

```
     /* compute r = a ^ p mod m (May be NULL for some implementations) */

int (*bn_mod_exp)(BIGNUM *r, BIGNUM *a, const BIGNUM *p,
          const BIGNUM *m, BN_CTX *ctx, BN_MONT_CTX *m_ctx);

     /* called at RSA_new */

int (*init)(RSA *rsa);

     /* called at RSA_free */

int (*finish)(RSA *rsa);

     /* RSA_FLAG_EXT_PKEY        - rsa_mod_exp is called for private key
      *                            operations, even if p,q,dmp1,dmq1,iqmp
      *                            are NULL
      * RSA_FLAG_SIGN_VER        - enable rsa_sign and rsa_verify
      * RSA_METHOD_FLAG_NO_CHECK - don't check pub/private match
      */

int flags;

char *app_data; /* ?? */

     /* sign. For backward compatibility, this is used only
      * if (flags & RSA_FLAG_SIGN_VER)
      */

int (*rsa_sign)(int type, unsigned char *m, unsigned int m_len,
           unsigned char *sigret, unsigned int *siglen, RSA *rsa);

     /* verify. For backward compatibility, this is used only
      * if (flags & RSA_FLAG_SIGN_VER)
      */

int (*rsa_verify)(int type, unsigned char *m, unsigned int m_len,
           unsigned char *sigbuf, unsigned int siglen, RSA *rsa);

 } RSA_METHOD;
```

RETURN VALUES

The `RSA_PKCS1_SSLeay()`, `RSA_PKCS1_RSAref()`, `RSA_PKCS1_null_method()`, `RSA_get_default_openssl_method()`, and `RSA_get_method()` functions return pointers to the respective RSA_METHODs.

The `RSA_set_default_openssl_method()` function returns no value.

The `RSA_set_method()` function selects `engine` as the engine that will be responsible for all operations using the structure `rsa`. If this function completes successfully, then the `rsa` structure will have its own functional reference of `engine`, so the caller should remember to free their own reference to `engine` when they are finished with it. An ENGINE's RSA_METHOD can be retrieved (or set) by the `ENGINE_get_RSA()` or `ENGINE_set_RSA()` functions.

The `RSA_new_method()` function returns NULL and sets an error code that can be obtained by using the `ERR_get_error()` function if the allocation fails. Otherwise it returns a pointer to the newly allocated structure.

HISTORY

The `RSA_new_method()` and `RSA_set_default_method()` functions appeared in SSLeay 0.8. The `RSA_get_default_method()`, `RSA_set_method()`, and `RSA_get_method()` functions as well as the `rsa_sign` and `rsa_verify` components of RSA_METHOD were added in OpenSSL 0.9.4.

The `RSA_set_default_openssl_method()` and `RSA_get_default_openssl_method()` functions replaced `RSA_set_default_method()` and `RSA_get_default_method()` respectively, and the `RSA_set_method()` and `RSA_new_method()` functions were altered to use `ENGINE`s rather than `DH_METHOD`s during development of OpenSSL 0.9.6.

SEE ALSO

Functions: *rsa*, *RSA_new*

RSA_sign

NAME

RSA_sign, RSA_verify – RSA signatures

SYNOPSIS

```
#include <openssl/rsa.h>

 int RSA_sign(

        int type, unsigned char *m, unsigned int m_len, unsigned char *sigret, unsigned
        int *siglen, RSA *rsa

);

int RSA_verify(

        int type, unsigned char *m, unsigned int m_len, unsigned char *sigbuf, unsigned
        int siglen, RSA *rsa

);
```

DESCRIPTION

The `RSA_sign()` function signs the message digest `m` of size `m_len` using the private key `rsa` as specified in PKCS #1 v2.0. It stores the signature in `sigret` and the signature size in `siglen`. The `sigret` must point to `RSA_size(rsa)` bytes of memory.

The `type` denotes the message digest algorithm that was used to generate `m`. It usually is one of `NID_sha1`, `NID_ripemd160` and `NID_md5`. See *objects* for details. If `type` is `NID_md5_sha1`, an SSL signature (MD5 and SHA1 message digests with PKCS #1 padding and no algorithm identifier) is created.

The `RSA_verify()` function verifies that the signature `sigbuf` of size `siglen` matches a given message digest `m` of size `m_len`. The `type` denotes the message digest algorithm that was used to generate the signature. The `rsa` is the signer's public key.

These functions conform to SSL, PKCS #1 v2.0.

RESTRICTIONS

Certain signatures with an improper algorithm identifier are accepted for compatibility with SSLeay 0.4.5.

RETURN VALUES

The `RSA_sign()` function returns 1 on success, 0 otherwise. The `RSA_verify()` function returns 1 on successful verification, 0 otherwise.

The error codes can be obtained by using the `ERR_get_error()` function.

HISTORY

The `RSA_sign()` and `RSA_verify()` functions are available in all versions of SSLeay and OpenSSL.

SEE ALSO

Functions: *err*, *objects*, *rsa*, *RSA_private_encrypt*, *RSA_public_decrypt*

RSA_sign_ASN1_OCTET_STRING

NAME

RSA_sign_ASN1_OCTET_STRING, RSA_verify_ASN1_OCTET_STRING – RSA signatures

SYNOPSIS

```
#include <openssl/rsa.h>

int RSA_sign_ASN1_OCTET_STRING(
        int dummy, unsigned char *m, unsigned int m_len, unsigned char *sigret, unsigned
        int *siglen, RSA *rsa
);

int RSA_verify_ASN1_OCTET_STRING(
        int dummy, unsigned char *m, unsigned int m_len, unsigned char *sigbuf, unsigned
        int siglen, RSA *rsa
);
```

DESCRIPTION

The `RSA_sign_ASN1_OCTET_STRING()` function signs the octet string `m` of size `m_len` using the private key `rsa` represented in DER using PKCS #1 padding. It stores the signature in `sigret` and the signature size in `siglen`. The `sigret` must point to `RSA_size(rsa)` bytes of memory.

The `dummy` is ignored.

The random number generator must be seeded prior to calling the `RSA_sign_ASN1_OCTET_STRING()` function.

The `RSA_verify_ASN1_OCTET_STRING()` function verifies that the signature `sigbuf` of size `siglen` is the DER representation of a given octet string `m` of size `m_len`. The `dummy` is ignored. The `rsa` is the signer's public key.

RESTRICTIONS

These functions serve no recognizable purpose.

RETURN VALUES

The `RSA_sign_ASN1_OCTET_STRING()` function returns 1 on success, 0 otherwise. The `RSA_verify_ASN1_OCTET_STRING()` function returns 1 on successful verification, 0 otherwise.

The error codes can be obtained by using the `ERR_get_error()` function.

HISTORY

The `RSA_sign_ASN1_OCTET_STRING()` and `RSA_verify_ASN1_OCTET_STRING()` functions were added in SSLeay 0.8.

SEE ALSO

Functions: *err*, *objects*, *rand*, *rsa*, *RSA_sign*, *RSA_verify*

RSA_size

NAME

RSA_size – Get RSA modulus size

SYNOPSIS

```
#include <openssl/rsa.h>
int RSA_size(
       RSA *rsa
);
```

DESCRIPTION

This function returns the RSA modulus size in bytes. It can be used to determine how much memory must be allocated for an RSA encrypted value.

The `rsa->n` must not be `NULL`.

RETURN VALUE

The size in bytes.

HISTORY

The `RSA_size()` function is available in all versions of SSLeay and OpenSSL.

SEE ALSO

Functions: *rsa*

rsautl

NAME

rsautl – RSA utility

SYNOPSIS

`openssl rsautl [-in` *filename*`] [-out` *filename*`] [-inkey` *filename*`] [-pubin] [-certin] [-sign] [-verify] [-encrypt] [-decrypt] [-pkcs] [-oaep] [-ssl] [-raw] [-hexdump] [-asn1parse]`

OPTIONS

`in` *filename*

Specifies the input filename to read data from or standard input if this option is not specified.

`out` *filename*

Specifies the output filename to write to or standard output by default.

`inkey file`

Input key file. By default it should be an RSA private key.

`pubin`

The input file is an RSA public key.

`certin`

The input is a certificate containing an RSA public key.

`sign`

Signs the input data and outputs the signed result. This requires and RSA private key.

`verify`

Verifies the input data and output the recovered data.

`encrypt`

Encrypts the input data using an RSA public key.

`decrypt`

Decrypts the input data using an RSA private key.

`pkcs`, `oaep`, `ssl`, `raw`

The padding to use: PKCS#1 v1.5 (the default), PKCS#1 OAEP, special padding used in SSL v2 backwards compatible handshakes, or no padding, respectively. For signatures, only `pkcs` and `raw` can be used.

`hexdump`

Hex dumps the output data.

`asn1parse`

Asn1parses the output data. This is useful when combined with the `verify` option.

DESCRIPTION

The `rsautl` command can be used to sign, verify, encrypt and decrypt data using the RSA algorithm.

NOTES

Because `rsautl` uses the RSA algorithm directly, it can only be used to sign or verify small pieces of data.

EXAMPLES

Sign some data using a private key:

```
openssl rsautl -sign -in file -inkey key.pem -out sig
```

Recover the signed data

```
openssl rsautl -verify -in sig -inkey key.pem
```

Examine the raw signed data:

```
openssl rsautl -verify -in file -inkey key.pem -raw -hexdump

0000 - 00 01 ff ff ff ff ff ff-ff ff ff ff ff ff ff ff   ................
0010 - ff ff ff ff ff ff ff ff-ff ff ff ff ff ff ff ff   ................
0020 - ff ff ff ff ff ff ff ff-ff ff ff ff ff ff ff ff   ................
0030 - ff ff ff ff ff ff ff ff-ff ff ff ff ff ff ff ff   ................
0040 - ff ff ff ff ff ff ff ff-ff ff ff ff ff ff ff ff   ................
0050 - ff ff ff ff ff ff ff ff-ff ff ff ff ff ff ff ff   ................
0060 - ff ff ff ff ff ff ff ff-ff ff ff ff ff ff ff ff   ................
0070 - ff ff ff ff 00 68 65 6c-6c 6f 20 77 6f 72 6c 64   .....hello world
```

The PKCS#1 block formatting is evident from this. If this was done using encrypt and decrypt the block would have been of type 2 (the second byte) and random padding data visible instead of the 0xff bytes.

It is possible to analyze the signature of certificates using this utility in conjunction with `asn1parse`. Consider the self-signed example in `certs/pca-cert.pem`. Running `asn1parse` yields the following:

```
openssl asn1parse -in pca-cert.pem

    0:d=0  hl=4 l= 742 cons: SEQUENCE
    4:d=1  hl=4 l= 591 cons:  SEQUENCE
    8:d=2  hl=2 l=   3 cons:   cont [ 0 ]
   10:d=3  hl=2 l=   1 prim:    INTEGER           :02
   13:d=2  hl=2 l=   1 prim:   INTEGER           :00
   16:d=2  hl=2 l=  13 cons:   SEQUENCE
   18:d=3  hl=2 l=   9 prim:    OBJECT            :md5WithRSAEncryption
   29:d=3  hl=2 l=   0 prim:    NULL
   31:d=2  hl=2 l=  92 cons:   SEQUENCE
   33:d=3  hl=2 l=  11 cons:    SET
   35:d=4  hl=2 l=   9 cons:     SEQUENCE
   37:d=5  hl=2 l=   3 prim:      OBJECT            :countryName
   42:d=5  hl=2 l=   2 prim:      PRINTABLESTRING   :AU
  ....
  599:d=1  hl=2 l=  13 cons:  SEQUENCE
  601:d=2  hl=2 l=   9 prim:   OBJECT            :md5WithRSAEncryption
  612:d=2  hl=2 l=   0 prim:   NULL
  614:d=1  hl=3 l= 129 prim:  BIT STRING
```

The final BIT STRING contains the actual signature. It can be extracted using the following command:

```
openssl asn1parse -in pca-cert.pem -out sig -noout -strparse 614
```

The certificate public key can be extracted using the following command:

```
openssl x509 -in test/testx509.pem -pubout -noout >pubkey.pem
```

The signature can be analyzed with:

```
openssl rsautl -in sig -verify -asn1parse -inkey pubkey.pem -pubin
   0:d=0  hl=2 l=  32 cons: SEQUENCE
   2:d=1  hl=2 l=  12 cons:  SEQUENCE
   4:d=2  hl=2 l=   8 prim:   OBJECT            :md5
  14:d=2  hl=2 l=   0 prim:   NULL
  16:d=1  hl=2 l=  16 prim:  OCTET STRING
0000 - f3 46 9e aa 1a 4a 73 c9-37 ea 93 00 48 25 08 b5 .F...Js.7...H%..
```

This is the parsed version of an ASN1 DigestInfo structure. The digest used was md5. The part of the certificate that was signed can be extracted with the following command:

```
openssl asn1parse -in pca-cert.pem -out tbs -noout -strparse 4
```

Its digest can be computed with the following command:

```
openssl md5 -c tbs
MD5(tbs)= f3:46:9e:aa:1a:4a:73:c9:37:ea:93:00:48:25:08:b5
```

This agrees with the recovered value above.

SEE ALSO

Commands: *dgst*, *rsa*, *genrsa*

s_client

NAME

s_client – SSL/TLS client program

SYNOPSIS

openssl s_client [-connect *host:port*] [-verify *depth*] [-cert *filename*] [-key *filename*] [-CApath *directory*] [-CAfile *filename*] [-reconnect] [-pause] [-showcerts] [-prexit] [-debug] [-nbio_test] [-state] [-nbio] [-crlf] [-ign_eof] [-quiet] [-ssl2] [-ssl3] [-tls1] [-no_ssl2] [-no_ssl3] [-no_tls1] [-bugs] [-cipher *cipherlist*] [-rand *filename*] [-engine *id*]

OPTIONS

connect *host:port*

Specifies the host and optional port to connect to. If not specified then an attempt is made to connect to the local host on port 4433.

cert *certname*

The certificate to use, if one is requested by the server. The default is not to use a certificate.

key *keyfile*

The private key to use. If not specified then the certificate file will be used.

verify *depth*

The verify depth to use. This specifies the maximum length of the server certificate chain and turns on server certificate verification. Currently the verify operation continues after errors so all the problems with a certificate chain can be seen. As a side effect the connection will never fail due to a server certificate verify failure.

CApath *directory*

The directory to use for server certificate verification. This directory must be in hash format. See `verify` for more information. These are also used when building the client certificate chain.

CAfile *file*

A file containing trusted certificates to use during server authentication and to use when attempting to build the client certificate chain.

reconnect

Reconnects to the same server 5 times using the same session ID. This can be used as a test that session caching is working.

pause

Pauses one second between each read and write call.

showcerts

Displays the whole server certificate chain. Normally only the server certificate is displayed.

prexit

Prints session information when the program exits. This will always attempt to print out information even if the connection fails. Normally information will only be printed out once if the connection succeeds. This option is useful because the cipher in use may be renegotiated or the connection may fail because a client certificate is required or is requested only after an attempt is made to access a certain URL. The output produced by this option is not always accurate because a connection might never have been established.

`state`

Prints out the SSL session states.

`debug`

Prints extensive debugging information including a hex dump of all traffic.

`nbio_test`

Tests nonblocking I/O

`nbio`

Turns on nonblocking I/O

`crlf`

Translates a line feed from the terminal into CR+LF as required by some servers.

`ign_eof`

Inhibits shutting down the connection when end-of-file is reached in the input.

`quiet`

Inhibits printing of session and certificate information. This implicitely turns on `ign_eof` as well.

`ssl2`, `ssl3`, `tls1`, `no_ssl2`, `no_ssl3`, `no_tls1`

These options disable the use of certain SSL or TLS protocols. By default the initial handshake uses a method which should be compatible with all servers and permit them to use SSL v3, SSL v2 or TLS as appropriate.

Unfortunately there are a lot of ancient and broken servers in use which cannot handle this technique and will fail to connect. Some servers only work if TLS is turned off with the `no_tls` option. Others will only support SSL v2 and may need the `ssl2` option.

`bugs`

There are several known bugs in SSL and TLS implementations. Adding this option enables various workarounds.

`cipher` *`cipherlist`*

Allows the cipher list sent by the client to be modified. Although the server determines which cipher suite is used it should take the first supported cipher in the list sent by the client. See the `ciphers` command for more information.

`rand` *`filename`*

A file or files containing random data used to seed the random number generator, or an EGD socket. (See *RAND_egd*.) Multiple files can be specified separated by an OS-dependent character. The separator is a semicolon (;) for MS-Windows, a comma (,) for OpenVMS, and a colon (:) for all others.

`engine` *`id`*

Specifying an engine (by its unique *id* string) will cause the `s_client` command to attempt to obtain a functional reference to the specified engine, thus initializing it if needed. The engine will then be set as the default for all available algorithms.

CONNECTED COMMANDS

If a connection is established with an SSL server then any data received from the server is displayed and any key presses will be sent to the server. When used interactively (which means neither `quiet` nor `ign_eof` have been given), the session will be renegotiated if the line begins with an `R`. If the line begins with a `Q` or if end-of-file is reached, the connection will be closed down.

DESCRIPTION

The `s_client` command implements a generic SSL/TLS client which connects to a remote host using SSL/TLS. It is a very useful diagnostic tool for SSL servers.

NOTES

The `s_client` command can be used to debug SSL servers. To connect to an SSL HTTP server, the following command would typically be used (https uses port 443):

```
openssl s_client -connect servername:443
```

If the connection succeeds then an HTTP command can be given such as `"GET /"` to retrieve a web page.

If the handshake fails then there are several possible causes. If it is nothing obvious, such as no client certificate, then the `bugs`, `ssl2`, `ssl3`, `tls1`, `no_ssl2`, `no_ssl3`, `no_tls1` options can be tried. You should try these options before submitting a bug report to an OpenSSL mailing list.

A frequent problem when attempting to get client certificates working is that a web client complains it has no certificates or gives an empty list to choose from. This is normally because the server is not sending the clients certificate authority in its acceptable CA list when it requests a certificate. By using `s_client` the CA list can be viewed and checked. However, some servers only request client authentication after a specific URL is requested. To obtain the list in this case it is necessary to use the `-prexit` option and send an HTTP request for an appropriate page.

If a certificate is specified on the command line using the `cert` option it will not be used unless the server specifically requests a client certificate. Therefore merely including a client certificate on the command line is no guarantee that the certificate works.

If there are problems verifying a server certificate then the `showcerts` option can be used to show the whole chain.

RESTRICTIONS

Because this program has a lot of options and also because some of the techniques used are rather old, the C source of `s_client` is hard to read and not a model of how things should be done. A typical SSL client program would be much simpler.

The `verify` option should exit if the server verification fails.

The `prexit` option should report information whenever a session is renegotiated.

SEE ALSO

Commands: *sess_id*, *s_server*, *ciphers*

s_server

NAME

s_server – SSL/TLS server program

SYNOPSIS

`openssl s_server [-accept` *port*`] [-context` *id*`] [-verify` *depth*`] [-Verify` *depth*`] [-cert` *filename*`] [-key` *keyfile*`] [-dcert` *filename*`] [-dkey` *keyfile*`] [-dHParam` *filename*`] [-nbio] [-nbio_test] [-crlf] [-debug] [-state] [-CApath` *directory*`] [-CAfile` *file*`] [-state] [-nocert] [-cipher` *cipherlist*`] [-quiet] [-no_tmp_rsa] [-ssl2] [-ssl3] [-tls1] [-no_ssl2] [-no_ssl3] [-no_tls1] [-no_dhe] [-bugs] [-hack] [-www] [-WWW] [-rand` *filename*`] [-engine` *id*`]`

OPTIONS

`accept` *port*

The TCP port to listen on for connections. If not specified 4433 is used.

`context` *id*

Sets the SSL context id. It can be given any string value. If this option is not present a default value will be used.

`cert` *certname*

The certificate to use. Most server's cipher suites require the use of a certificate and some require a certificate with a certain public key type. For example, the DSS cipher suites require a certificate containing a DSS (DSA) key. If not specified then the filename `server.pem` will be used.

`key` *keyfile*

The private key to use. If not specified then the certificate file will be used.

`dcert` *filename*

Specifies an additional certificate and private key. These behave in the same manner as the `cert` and `key` options except there is no default if they are not specified (no additional certificate and key is used). Some cipher suites require a certificate containing a key of a certain type. Some cipher suites need a certificate carrying an RSA key and some a DSS (DSA) key. By using RSA and DSS certificates and keys, a server can support clients which only support RSA or DSS cipher suites by using an appropriate certificate.

`nocert`

If this option is set then no certificate is used. This restricts the cipher suites available to the anonymous ones (currently just anonymous DH).

`dHParam` *filename*

The DH parameter file to use. The ephemeral DH cipher suites generate keys using a set of DH parameters. If not specified then an attempt is made to load the parameters from the server certificate file. If this fails then a static set of parameters hard coded into the `s_server` program will be used.

`no_dhe`

If this option is set then no DH parameters will be loaded effectively disabling the ephemeral DH cipher suites.

`no_tmp_rsa`

Certain export cipher suites sometimes use a temporary RSA key, this option disables temporary RSA key generation.

`verify` *depth*, `Verify` *depth*

The verify depth to use. This specifies the maximum length of the client certificate chain and makes the server request a certificate from the client. With the `verify` option a certificate is requested but the client does not have to send one. With the `Verify` option the client must supply a certificate or an error occurs.

`CApath` *directory*

The directory to use for client certificate verification. This directory must be in hash format. See `verify` for more information. These are also used when building the server certificate chain.

`CAfile` *file*

A file containing trusted certificates to use during client authentication and to use when attempting to build the server certificate chain. The list is also used in the list of acceptable client CAs passed to the client when a certificate is requested.

`state`

Prints out the SSL session states.

`debug`

Prints extensive debugging information including a hex dump of all traffic.

`nbio_test`

Tests non-blocking I/O

`nbio`

Turns on non-blocking I/O

`crlf`

Translates a line feed from the terminal into CR+LF.

`quiet`

Inhibits printing of session and certificate information.

`ssl2`, `ssl3`, `tls1`, `no_ssl2`, `no_ssl3`, `no_tls1`

Disables the use of certain SSL or TLS protocols. By default the initial handshake uses a method which should be compatible with all servers and permit them to use SSL v3, SSL v2 or TLS as appropriate.

`bugs`

There are several known bugs in SSL and TLS implementations. Adding this option enables various workarounds.

`hack`

Enables a further workaround for some early Netscape SSL code.

`cipher` *cipherlist*

Allows the cipher list used by the server to be modified. When the client sends a list of supported ciphers the first client cipher also included in the server list is used. Because the client specifies the preference order, the order of the server cipherlist is irrelevant. See the `ciphers` command for more information.

`www`

Sends a status message back to the client when it connects. This includes lots of information about the ciphers used and various session parameters. The output is in HTML format so this option will normally be used with a web browser.

`WWW`

Emulates a simple web server. Pages will be resolved relative to the current directory. For example, if the URL `https://myhost/page.html` is requested, the file `./page.html` will be loaded.

`rand` *`filename`*

A file or files containing random data used to seed the random number generator, or an EGD socket. (See *RAND_egd*.) Multiple files can be specified separated by an OS-dependent character. The separator is a semicolon (;) for MS-Windows, a comma (,) for OpenVMS, and a colon (:) for all others.

`engine` *`id`*

Specifying an engine (by its unique `id` string) will cause `s_server` to attempt to obtain a functional reference to the specified engine, thus initializing it if needed. The engine will then be set as the default for all available algorithms.

Connected Commands

If a connection request is established with an SSL client and neither the `www` nor the `WWW` option has been used then normally any data received from the client is displayed and any key presses will be sent to the client.

Certain single letter commands are also recognized which perform special operations. These are:

`q`

Ends the current SSL connection but still accept new connections.

`Q`

Ends the current SSL connection and exit.

`r`

Renegotiates the SSL session.

`R`

Renegotiates the SSL session and request a client certificate.

`P`

Sends some plain text down the underlying TCP connection: this should cause the client to disconnect due to a protocol violation.

`S`

Prints out some session cache status information.

DESCRIPTION

The `s_server` command implements a generic SSL/TLS server which listens for connections on a given port using SSL/TLS.

NOTES

The `s_server` command can be used to debug SSL clients. To accept connections from a web browser the following command can be used:

```
openssl s_server -accept 443 -www
```

Most web browsers (in particular Netscape and MSIE) only support RSA cipher suites, so they cannot connect to servers which do not use a certificate carrying an RSA key or a version of OpenSSL with RSA disabled.

Although specifying an empty list of CAs when requesting a client certificate is strictly speaking a protocol violation, some SSL clients interpret this to mean any CA is acceptable. This is useful for debugging purposes.

The session parameters can be printed out using the `sess_id` program.

RESTRICTIONS

Because this program has a lot of options and also because some of the techniques used are rather old, the C source of `s_server` is rather hard to read and not a model of how things should be done. A typical SSL server program would be much simpler.

The output of common ciphers is wrong. It only gives the list of ciphers that OpenSSL recognizes and the client supports.

There should be a way for the `s_server` program to print out details of any unknown cipher suites a client says it supports.

SEE ALSO

Commands: *sess_id*, *s_client*, *ciphers*

sess_id

NAME

sess_id – SSL/TLS session handling utility

SYNOPSIS

openssl sess_id [-inform PEM|DER] [-outform PEM|DER] [-in *filename*] [-out *filename*] [-text] [-noout] [-contextID]

OPTIONS

inform DER|PEM

Specifies the input format. The DER option uses an ASN1 DER encoded format containing session details. The precise format can vary from one version to the next. The PEM form is the default format. It consists of the DER format base64 encoded with additional header and footer lines.

outform DER|PEM

Specifies the output format. The options have the same meaning as the inform option.

in *filename*

Specifies the input filename to read session information from or standard input by default.

out *filename*

Specifies the output filename to write session information to or standard output if this option is not specified.

text

Prints out the various public or private key components in plain text in addition to the encoded version.

cert

If a certificate is present in the session it will be output using this option. If the text option is also present then it will be printed out in text form.

noout

Prevents output of the encoded version of the session.

context *ID*

Sets the session id so the output session information uses the supplied ID. The ID can be any string of characters. This option usually is not used.

DESCRIPTION

The sess_id processes the encoded version of the SSL session structure and optionally prints out SSL session details, such as the SSL session master key, in human readable format. Since this is a diagnostic tool that needs some knowledge of the SSL protocol to use properly, most users will not need to use it.

NOTES

The PEM encoded session format uses the following header and footer lines:

```
-----BEGIN SSL SESSION PARAMETERS-----
-----END SSL SESSION PARAMETERS-----
```

Since the SSL session output contains the master key it is possible to read the contents of an encrypted session using this information. Therefore appropriate security precautions should be taken if the information is being output by a real application. This is, however, strongly discouraged and should only be used for debugging purposes.

RESTRICTIONS

The cipher and start time should be printed out in human readable form.

EXAMPLES

An example of typical output follows:

```
SSL-Session:
    Protocol  : TLSv1
    Cipher    : 0016
    Session-ID: 871E62626C554CE95488823752CBD5F3673A3EF3DCE9C67BD916C809914B40ED
    Session-ID-ctx: 01000000
    Master-Key: A7CEFC571974BE02CAC305269DC59F76EA9F0B180CB6642697A68251F2D2BB57
                E51DBBB4C7885573192AE9AEE220FACD
    Key-Arg   : None
    Start Time: 948459261
    Timeout   : 300 (sec)
    Verify return code 0 (ok)
```

These are described below in more detail.

`Protocol`

The protocol in use - TLSv1, SSLv3 or SSLv2.

`Cipher`

The cipher used. This is the actual raw SSL or TLS cipher code. See the SSL or TLS specifications for more information.

`Session-ID`

The SSL session ID in hex format.

`Session-ID-ctx`

The session ID context in hex format.

`Master-Key`

The SSL session master key.

`Key-Arg`

The key argument. This is only used in SSL v2.

`Start Time`

The session start time, represented as an integer in standard UNIX format.

`Timeout`

The timeout in seconds.

`Verify return code`

The return code when an SSL client certificate is verified.

SEE ALSO

Commands: *ciphers*, *s_server*

SHA

NAME

SHA: SHA1, SHA1_Init, SHA1_Update, SHA1_Final – Secure Hash Algorithm

SYNOPSIS

```
#include <openssl/sha.h>
unsigned char *SHA1(
        const unsigned char *d, unsigned long n, unsigned char *md
);
void SHA1_Init(
        SHA_CTX *c
);
void SHA1_Update(
        SHA_CTX *c, const void *data, unsigned long len
);
void SHA1_Final(
        unsigned char *md, SHA_CTX *c
);
```

DESCRIPTION

SHA-1 (Secure Hash Algorithm) is a cryptographic hash function with a 160-bit output.

The `SHA1()` function computes the SHA-1 message digest of the `n` bytes at `d` and places it in `md` (which must have space for `SHA_DIGEST_LENGTH == 20 bytes of output`). If `md` is NULL, the digest is placed in a static array.

The following functions can be used if the message is not completely stored in memory:

The `SHA1_Init()` function initializes a `SHA_CTX` structure.

The `SHA1_Update()` function can be called repeatedly with chunks of the message to be hashed (`len` bytes at `data`).

The `SHA1_Final()` function places the message digest in `md`, which must have space for `SHA_DIGEST_LENGTH == 20 bytes of output`, and erases the `SHA_CTX`.

Applications should use the higher level functions, such as `EVP_DigestInit()`, instead of calling the hash functions directly.

The predecessor of SHA-1, SHA, is also implemented, but it should be used only when backward compatibility is required.

RETURN VALUES

The `SHA1()` function returns a pointer to the hash value.

The `SHA1_Init()`, `SHA1_Update()`, and `SHA1_Final()` functions do not return values.

HISTORY

The `SHA1()`, `SHA1_Init()`, `SHA1_Update()`, and `SHA1_Final()` functions are available in all versions of SSLeay and OpenSSL.

SHA conforms to US Federal Information Processing Standard FIPS PUB 180 (Secure Hash Standard). SHA-1 conforms to US Federal Information Processing Standard FIPS PUB 180-1 (Secure Hash Standard), ANSI X9.30

SEE ALSO

Functions: *ripemd160*, *hmac*, *EVP_DigestInit*

smime

NAME

smime – S/MIME utility

SYNOPSIS

openssl smime [-encrypt] [-decrypt] [-sign] [-verify] [-pk7out] [-in *filename*] [-inform SMIME|PEM|DER] [-des] [-out *filename*] [-outform SMIME|PEM|DER] [-des3] [-rc2-40] [-rc2-64] [-rc2-128] [-nointern] [-noverify] [-nochain] [-nosigs] [-nocerts] [-noattr] [-binary] [-nodetach] [-certfile *filename*] [-signer *filename*] [-recip *filename*] [-passin *arg*] [-inkey *filename*] [-content *filename*] [-to *addr*] [-fromad] [-subject s] [-text] [-CAfile *filename*] [-CApath *dir*] [-rand *filename*] [-cert.pem ...]

OPTIONS

There are five options that set the type of operation to be performed. The meaning of the other options varies according to the operation type.

encrypt

Encrypts mail for the given recipient certificates. Input file is the message to be encrypted. The output file is the encrypted mail in MIME format.

decrypt

Decrypts mail using the supplied certificate and private key. Expects an encrypted mail message in MIME format for the input file. The decrypted mail is written to the output file.

sign

Signs mail using the supplied certificate and private key. Input file is the message to be signed. The signed message in MIME format is written to the output file.

verify

Verifies signed mail. Expects a signed mail message on input, and outputs the signed data. Both clear text and opaque signing is supported.

pk7out

Takes an input message and writes out a PEM encoded PKCS#7 structure.

in *filename*

The input message to be encrypted or signed, or the MIME message to be decrypted or verified.

inform SMIME|PEM|DER

Specifies the input format for the PKCS#7 structure. The default is SMIME which reads an S/MIME format message. The PEM and the DER format change this to expect PEM and DER format PKCS#7 structures instead. This only affects the input format of the PKCS#7 structure. If no PKCS#7 structure is input, such as encrypt or sign, this option has no effect.

out *filename*

The message text that has been decrypted or verified or the output MIME format message that has been signed or verified.

`outform SMIME|PEM|DER`

Specifies the output format for the PKCS#7 structure. The default is `SMIME` which writes an S/MIME format message. The `PEM` and `DER` format change this to write `PEM` and `DER` format PKCS#7 structures instead. This only affects the output format of the PKCS#7 structure. If no PKCS#7 structure is output, such as `verify` or `decrypt`, this option has no effect.

`content` *`filename`*

Specifies a file containing the detached content. This is only useful with the `verify` option. This is only usable if the PKCS#7 structure is using the detached signature form where the content is not included. This option will override any content if the input format is S/MIME and it uses the multipart/signed MIME content type.

`text`

Adds plain text (text/plain) MIME headers to the supplied message if encrypting or signing. If decrypting or verifying it strips off text headers. If the decrypted or verified message is not of MIME type text/plain then an error occurs.

`CAfile` *`file`*

A file containing trusted CA certificates. It is only used with the `verify` option.

`CApath` *`dir`*

A directory containing trusted CA certificates. It is only used with the `verify` option. This directory must be a standard certificate directory, meaning a hash of each subject name (using `x509 -hash`) should be linked to each certificate.

`des des3 rc2-40 rc2-64 rc2-128`

The encryption algorithm to use. DES (56 bits), triple DES (168 bits) or 40, 64 or 128 bit RC2, respectively. If not specified, 40-bit RC2 is used. These are used only with the `encrypt` option.

`nointern`

When verifying a message, certificates (if any) included in the message are searched for the signing certificate. With this option only the certificates specified in the `certfile` option are used. The supplied certificates can still be used as untrusted CAs however.

`noverify`

Does not verify the signers certificate of a signed message.

`nochain`

Does not perform chain verification of signers certificates. That is, it does not use the certificates in the signed message as untrusted CAs.

`nosigs`

Does not try to verify the signatures on the message.

`nocerts`

When signing a message, the signer's certificate is usually included. With this option the signer's certificate is excluded. This will reduce the size of the signed message, but the verifier must have a copy of the signer's certificate available locally (passed using the `certfile` option, for example).

`noattr`

When a message is signed, a set of attributes is included, such as the signing time and supported symmetric algorithms. With this option they are not included.

`binary`

Usually the input message is converted to canonical format, which is effectively using CR and LF as end-of-line, as required by the S/MIME specification. With this option no translation occurs. This is useful when handling binary data which may not be in MIME format.

`nodetach`

Uses opaque signing when signing a message. This form is more resistant to translation by mail relays, but it cannot be read by mail agents that do not support S/MIME. Without this option cleartext signing with the MIME type multipart/signed is used.

`certfile` *`filename`*

Allows additional certificates to be specified. When signing these will be included with the message. When verifying, these will be searched for the signer's certificates. The certificates should be in `PEM` format.

`signer` *`filename`*

The signer's certificate when signing a message. If a message is being verified then the signer's certificates will be written to this file if the verification was successful.

`recip` *`filename`*

The recipient's certificate when decrypting a message. This certificate must match one of the recipient's of the message or an error occurs.

`inkey` *`filename`*

The private key to use when signing or decrypting. This must match the corresponding certificate. If this option is not specified then the private key must be included in the certificate file specified with the `recip` or the `signer` option.

`passin` *`arg`*

The private key password source. For more information about the format of `arg`, see the Pass Phrase Arguments section in *openssl*.

`rand` *`filename`*

A file or files containing random data used to seed the random number generator, or an EGD socket. (See *RAND_egd*.) Multiple files can be separated by an OS-dependent character. The separator is a semicolon (;) for MS-Windows, a comma (,) for OpenVMS, and a colon (:) for all others.

`cert.pem...`

One or more certificates of message recipients, used when encrypting a message.

`to`, `from`, `subject`

The relevant mail headers. These are included outside the signed portion of a message so they may be included manually. If signing, then many S/MIME mail clients check that the signer's certificate email address matches that specified in the `From: address`.

DESCRIPTION

The `smime` command handles S/MIME mail. It can encrypt, decrypt, sign and verify S/MIME messages.

NOTES

The MIME message must be sent without any blank lines between the headers and the output. Some mail programs will automatically add a blank line. Piping the mail directly to sendmail is one way to achieve the correct format.

The supplied message to be signed or encrypted must include the necessary MIME headers or many S/MIME clients will not display it properly (if at all). You can use the `text` option to automatically add plain text headers.

A signed and encrypted message is one where a signed message is then encrypted. This can be produced by encrypting an already signed message.

This version of the program only allows one signer per message, but it will verify multiple signers on received messages. Some S/MIME clients fail if a message contains multiple signers. It is possible to sign messages in parallel by signing an already signed message.

The options `encrypt` and `decrypt` reflect common usage in S/MIME clients. These process PKCS#7 enveloped data. The PKCS#7 encrypted data is used for other purposes.

RESTRICTIONS

The MIME parser is not very clever. It seems to handle most messages, but it may fail on others.

The code will only write out the signer's certificate to a file. If the signer has a separate encryption certificate this must be manually extracted. There should be some heuristic that determines the correct encryption certificate.

Ideally a certificate database should be maintained for each email address.

The code does not take note of the permitted symmetric encryption algorithms as supplied in the SMIMECapabilities signed attribute. This means the user has to manually include the correct encryption algorithm. It should store the list of permitted ciphers in a database and only use those.

No revocation checking is done on the signer's certificate.

The code can only handle S/MIME v2 messages. The more complex S/MIME v3 structures may cause parsing errors.

EXIT STATUS

0

The operation was completely successfully.

1

An error occurred parsing the command options.

2

One of the input files could not be read.

3

An error occurred creating the PKCS#7 file or when reading the MIME message.

4

An error occurred decrypting or verifying the message.

5

The message was verified correctly but an error occurred writing out the signers certificates.

EXAMPLES

Create a cleartext signed message:

```
openssl smime -sign -in message.txt -text -out mail.msg \
-signer mycert.pem
```

Create an opaque signed message:

```
openssl smime -sign -in message.txt -text -out mail.msg -nodetach \
-signer mycert.pem
```

Create a signed message, include some additional certificates and read the private key from another file:

```
openssl smime -sign -in in.txt -text -out mail.msg \
-signer mycert.pem -inkey mykey.pem -certfile mycerts.pem
```

Send a signed message under UNIX directly to sendmail, including headers:

```
openssl smime -sign -in in.txt -text -signer mycert.pem \
-from steve@openssl.org -to someone@somewhere \
-subject "Signed message" | sendmail someone@somewhere
```

Verify a message and extract the signer's certificate if successful:

```
openssl smime -verify -in mail.msg -signer user.pem -out signedtext.txt
```

Send encrypted mail using triple DES:

```
openssl smime -encrypt -in in.txt -from steve@openssl.org \
-to someone@somewhere -subject "Encrypted message" \
-des3 user.pem -out mail.msg
```

Sign and encrypt mail:

```
openssl smime -sign -in ml.txt -signer my.pem -text \
| openssl smime -encrypt -out mail.msg \
-from steve@openssl.org -to someone@somewhere \
-subject "Signed and Encrypted message" -des3 user.pem
```

Notice that the encryption command does not include the `text` option because the message being encrypted already has MIME headers.

Decrypt mail:

```
openssl smime -decrypt -in mail.msg -recip mycert.pem -inkey key.pem
```

The output from Netscape form signing is a PKCS#7 structure with the detached signature format. You can use this program to verify the signature by line wrapping the base64 encoded structure and surrounding it with the following lines:

```
 -----BEGIN PKCS7----
 -----END PKCS7----
```

You should then use the following command:

```
openssl smime -verify -inform PEM -in signature.pem -content content.txt
```

Alternatively, you can base64 decode the signature and use the following command:

```
openssl smime -verify -inform DER -in signature.der -content content.txt
```

speed

NAME

speed – Tests library performance

SYNOPSIS

openssl speed [-engine *id*] [-md2] [-mdc2] [-md5] [-hmac] [-sha1] [-rmd160] [-idea-cbc] [-rc2-cbc] [-rc5-cbc] [-bf-cbc] [-des-cbc] [-des-ede3] [-rc4] [-rsa512] [-rsa1024] [-rsa2048] [-rsa4096] [-dsa512] [-dsa1024] [-dsa2048] [-idea] [-rc2] [-des] [-rsa] [-blowfish]

OPTIONS

engine *id*

Specifying an engine (by its unique id string) will cause the speed command to attempt to obtain a functional reference to the specified engine, thus initialiszing it if needed. The engine will then be set as the default for all available algorithms.

[zero or more test algorithms]

If any options are given, the speed command tests those algorithms. Otherwise all of the above are tested.

DESCRIPTION

The speed command tests the performance of cryptographic algorithms.

spkac

NAME

spkac – SPKAC printing and generating utility

SYNOPSIS

openssl spkac [-in *filename*] [-out *filename*] [-key *keyfile*] [-passin *arg*] [-challenge *string*] [-pubkey] [-spkac *spkacname*] [-spksect *section*] [-noout] [-verify]

OPTIONS

in *filename*

Specifies the input filename to read from or standard input if this option is not specified. Ignored if the key option is used.

out *filename*

Specifies the output filename to write to or standard output by default.

key *keyfile*

Creates an SPKAC file using the private key in keyfile. The in, noout, spksect and verify options are ignored if present.

passin *password*

The input file password source. For more information about the format of arg, see the Pass Phrase Arguments section in *openssl*.

challenge *string*

Specifies the challenge string if an SPKAC is being created.

spkac *spkacname*

Allows an alternative name from the variable containing the SPKAC. The default is SPKAC. This option affects both generated and input SPKAC files.

spksect *section*

Allows an alternative name from the section containing the SPKAC. The default is the default section.

noout

Does not output the text version of the SPKAC (not used if an SPKAC is being created).

pubkey

Outputs the public key of an SPKAC (not used if an SPKAC is being created).

verify

Verifies the digital signature on the supplied SPKAC.

DESCRIPTION

The spkac command processes Netscape signed public key and challenge (SPKAC) files. It can print out their contents, verify the signature and produce its own SPKACs from a supplied private key.

NOTES

A created SPKAC with suitable DN components appended can be fed into the `ca` utility.

SPKACs are typically generated by Netscape when a form is submitted containing the `KEYGEN` tag as part of the certificate enrollment process.

The challenge string permits a primitive form of proof of possession of private key. By checking the SPKAC signature and a random challenge string some guarantee is given that the user knows the private key corresponding to the public key being certified. This is important in some applications. Without this it is possible for a previous SPKAC to be used in a replay attack.

EXAMPLES

Print out the contents of an SPKAC:

```
openssl spkac -in spkac.cnf
```

Verify the signature of an SPKAC:

```
openssl spkac -in spkac.cnf -noout -verify
```

Create an SPKAC using the challenge string "hello":

```
openssl spkac -key key.pem -challenge hello -out spkac.cnf
```

Example of an SPKAC (long lines split up for clarity):

```
SPKAC=MIG5MGUwXDANBgkqhkiG9w0BAQEFAANLADBIAkEA1cCoq2Wa3Ixs47uI7F\
PVwHVIPDx5yso105Y6zpozam135a8R0CpoRvkkigIyXfcCjiVi5oWk+6FfPaD03u\
PFoQIDAQABFgVoZWxsbzANBgkqhkiG9w0BAQQFAANBAFpQtY/FojdwkJh1bEIYuc\
2EeM2KHTWPEepWYeawvHD0gQ3DngSC75YCWnnDdq+NQ3F+X4deMx9AaEglZtULwV\
4=
```

SEE ALSO

Commands: *ca*

ssl

NAME

ssl, SSL – OpenSSL SSL/TLS library

DESCRIPTION

The OpenSSL `ssl` library implements the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) protocols. It provides a rich API which is documented here.

At first, the library must be initialized; see *SSL_library_init*.

Then an SSL_CTX object is created as a framework to establish TLS/SSL enabled connections (see *SSL_CTX_new*). Various options regarding certificates, algorithms, etc. can be set in this object.

When a network connection has been created, it can be assigned to an SSL object. After the SSL object has been created using `SSL_new()`, `SSL_set_fd()` or `SSL_set_bio()`, it can be used to associate the network connection with the object.

Then the TLS/SSL handshake is performed using `SSL_accept()` or `SSL_connect()` respectively. The `SSL_read()` and `SSL_write()` functions are used to read and write data on the TLS/SSL connection. The `SSL_shutdown()` function can be used to shut down the TLS/SSL connection.

DATA STRUCTURES

The OpenSSL ssl library functions deal with the following data structures:

`SSL_METHOD` (SSL Method)

A dispatch structure describing the internal `ssl` library methods and functions which implement the various protocol versions (SSLv1, SSLv2 and TLSv1). It is s needed to create an `SSL_CTX`.

`SSL_CIPHER` (SSL Cipher)

A structure that holds the algorithm information for a particular cipher which are a core part of the SSL/TLS protocol. The available ciphers are configured on a `SSL_CTX` basis and the actually used ones are then part of the `SSL_SESSION`.

`SSL_CTX` (SSL Context)

The global context structure that is created by a server or client once per program life-time. It holds default values for the `SSL` structures which are later created for the connections.

`SSL_SESSION` (SSL Session)

A structure containing the current TLS/SSL session details for a connection: `SSL_CIPHER`s, client and server certificates, keys, etc.

`SSL` (SSL Connection)

The main SSL/TLS structure that is created by a server or client per established connection. This is the core structure in the SSL API. Under run-time the application usually deals with this structure which has links to other structures.

HEADER FILES

The OpenSSL `ssl` library provides the following C header files containing the prototypes for the data structures and functions:

`ssl.h`

The common header file for the SSL/TLS API. Include it in your program to make the API of the `ssl` library available. It internally includes private SSL headers and headers from the crypto library. Whenever you need details on the internals of the SSL API, look inside this header file.

`ssl2.h`

The sub header file dealing with the SSLv2 protocol only. Usually you do not have to include it because it is already included by ssl.h.

`ssl3.h`

The sub header file dealing with the SSLv3 protocol only. Usually you do not have to include it because it is already included by ssl.h.

`ssl23.h`

The sub header file dealing with the combined use of the SSLv2 and SSLv3 protocols. Usually you do not have to include it because it is already included by ssl.h.

`tls1.h`

The sub header file dealing with the TLSv1 protocol only. Usually you do not have to include it because it is already included by ssl.h.

API FUNCTIONS

The OpenSSL `ssl` library exports 214 API functions. They are documented in the following sections.

Dealing with Protocol Methods

The API functions that deal with the SSL/TLS protocol methods defined in `SSL_METHOD` structures are described in the following list:

SSL_METHOD *`SSLv2_client_ method`(void);

Constructor for the SSLv2 SSL_METHOD structure for a dedicated client.

SSL_METHOD *`SSLv2_server_ method`(void);

Constructor for the SSLv2 SSL_METHOD structure for a dedicated server.

SSL_METHOD *`SSLv2_method`(void);

Constructor for the SSLv2 SSL_METHOD structure for combined client and server.

SSL_METHOD *`SSLv3_client_ method`(void);

Constructor for the SSLv3 SSL_METHOD structure for a dedicated client.

SSL_METHOD *`SSLv3_server_ method`(void);

Constructor for the SSLv3 SSL_METHOD structure for a dedicated server.

SSL_METHOD *`SSLv3_method`(void);

Constructor for the SSLv3 SSL_METHOD structure for combined client and server.

SSL_METHOD *`TLSv1_client_ method`(void);

Constructor for the TLSv1 SSL_METHOD structure for a dedicated client.

SSL_METHOD *`TLSv1_server_ method`(void);

Constructor for the TLSv1 SSL_METHOD structure for a dedicated server.

SSL_METHOD *TLSv1_method(void);

Constructor for the TLSv1 SSL_METHOD structure for combined client and server.

Dealing with Ciphers

The API functions that deal with the SSL/TLS ciphers defined in SSL_CIPHER structures are described in the following list:

char *SSL_CIPHER_description (SSL_CIPHER *cipher, char *buf, int len);

Write a string to buf (with a maximum size of len) containing a human readable description of cipher. Returns buf.

int SSL_CIPHER_get_bits (SSL_CIPHER *cipher, int *alg_bits);

Determine the number of bits in cipher. Because of export crippled ciphers there are two bits: The bits the algorithm supports in general (stored to alg_bits) and the bits which are actually used (the return value).

const char *SSL_CIPHER_get_name (SSL_CIPHER *cipher);

Return the internal name of cipher s a string. These are the various strings defined by the SSL2_TXT_xxx, SSL3_TXT_xxxand TLS1_TXT_xxxdefinitions in the header files.

char *SSL_CIPHER_get_version (SSL_CIPHER *cipher);

Returns a string such as TLSv1/SSLv3 or SSLv2 which indicates the SSL/TLS protocol version to which cipher belongs (i.e. where it was defined in the specification the first time).

Dealing with Protocol Contexts

The API functions that deal with the SSL/TLS protocol context defined in the SSL_CTX structure are described in the following list:

- int SSL_CTX_add_client_CA(SSL_CTX *ctx, X509 *x);
- long SSL_CTX_add_extra_chain_cert(SSL_CTX *ctx, X509 *x509);
- int SSL_CTX_add_session(SSL_CTX *ctx, SSL_SESSION *c);
- int SSL_CTX_check_private_key(SSL_CTX *ctx);
- long SSL_CTX_ctrl(SSL_CTX *ctx, int cmd, long larg, char *parg);
- void SSL_CTX_flush_sessions(SSL_CTX *s, long t);
- void SSL_CTX_free(SSL_CTX *a);
- char *SSL_CTX_get_app_data(SSL_CTX *ctx);
- X509_STORE *SSL_CTX_get_cert_store(SSL_CTX *ctx);
- STACK *SSL_CTX_get_client_CA_list(SSL_CTX *ctx);
- int (*SSL_CTX_get_client_cert_cb(SSL_CTX *ctx))(SSL *ssl, X509 **x509, EVP_PKEY **pkey);
- char *SSL_CTX_get_ex_data(SSL_CTX *s, int idx);
- int SSL_CTX_get_ex_new_index(long argl, char *argp, int (*new_func);(void), int (*dup_func)(void), void (*free_func)(void));
- void (*SSL_CTX_get_info_callback(SSL_CTX *ctx))(SSL *ssl, int cb, int ret);

- int SSL_CTX_get_quiet_shutdown(SSL_CTX *ctx);
- int SSL_CTX_get_session_cache_mode(SSL_CTX *ctx);
- long SSL_CTX_get_timeout(SSL_CTX *ctx);
- int (*SSL_CTX_get_verify_callback(SSL_CTX *ctx))(int ok, X509_STORE_CTX *ctx);
- int SSL_CTX_get_verify_mode(SSL_CTX *ctx);
- int SSL_CTX_load_verify_locations(SSL_CTX *ctx, char *CAfile, char *CApath);
- long SSL_CTX_need_tmp_RSA(SSL_CTX *ctx);
- SSL_CTX *SSL_CTX_new(SSL_METHOD *meth);
- int SSL_CTX_remove_session(SSL_CTX *ctx, SSL_SESSION *c);
- int SSL_CTX_sess_accept(SSL_CTX *ctx);
- int SSL_CTX_sess_accept_good(SSL_CTX *ctx);
- int SSL_CTX_sess_accept_renegotiate(SSL_CTX *ctx);
- int SSL_CTX_sess_cache_full(SSL_CTX *ctx);
- int SSL_CTX_sess_cb_hits(SSL_CTX *ctx);
- int SSL_CTX_sess_connect(SSL_CTX *ctx);
- int SSL_CTX_sess_connect_good(SSL_CTX *ctx);
- int SSL_CTX_sess_connect_renegotiate(SSL_CTX *ctx);
- int SSL_CTX_sess_get_cache_size(SSL_CTX *ctx);
- SSL_SESSION *(*SSL_CTX_sess_get_get_cb(SSL_CTX *ctx))(SSL *ssl, unsigned char *data, int len, int *copy);
- int (*SSL_CTX_sess_get_new_cb(SSL_CTX *ctx)(SSL *ssl, SSL_SESSION *sess);
- void (*SSL_CTX_sess_get_remove_cb(SSL_CTX *ctx)(SSL_CTX *ctx, SSL_SESSION *sess);
- int SSL_CTX_sess_hits(SSL_CTX *ctx);
- int SSL_CTX_sess_misses(SSL_CTX *ctx);
- int SSL_CTX_sess_number(SSL_CTX *ctx);
- void SSL_CTX_sess_set_cache_size(SSL_CTX *ctx,t);
- void SSL_CTX_sess_set_get_cb(SSL_CTX *ctx, SSL_SESSION *(*cb)(SSL *ssl, unsigned char *data, int len, int *copy));
- void SSL_CTX_sess_set_new_cb(SSL_CTX *ctx, int (*cb)(SSL *ssl, SSL_SESSION *sess));
- void SSL_CTX_sess_set_remove_cb(SSL_CTX *ctx, void (*cb)(SSL_CTX *ctx, SSL_SESSION *sess));
- int SSL_CTX_sess_timeouts(SSL_CTX *ctx);
- LHASH *SSL_CTX_sessions(SSL_CTX *ctx);
- void SSL_CTX_set_app_data(SSL_CTX *ctx, void *arg);
- void SSL_CTX_set_cert_store(SSL_CTX *ctx, X509_STORE *cs);
- void SSL_CTX_set_cert_verify_cb(SSL_CTX *ctx, int (*cb)(SSL_CTX *), char *arg);
- int SSL_CTX_set_cipher_list(SSL_CTX *ctx, char *str);

- void SSL_CTX_set_client_CA_list(SSL_CTX *ctx, STACK *list);
- void SSL_CTX_set_client_cert_cb(SSL_CTX *ctx, int (*cb)(SSL *ssl, X509 **x509, EVP_PKEY **pkey));
- void SSL_CTX_set_default_passwd_cb(SSL_CTX *ctx, int (*cb);(void));
- void SSL_CTX_set_default_read_ahead(SSL_CTX *ctx, int m);
- int SSL_CTX_set_default_verify_paths(SSL_CTX *ctx);
- int SSL_CTX_set_ex_data(SSL_CTX *s, int idx, char *arg);
- void SSL_CTX_set_info_callback(SSL_CTX *ctx, void (*cb)(SSL *ssl, int cb, int ret));
- void SSL_CTX_set_options(SSL_CTX *ctx, unsigned long op);
- void SSL_CTX_set_quiet_shutdown(SSL_CTX *ctx, int mode);
- void SSL_CTX_set_session_cache_mode(SSL_CTX *ctx, int mode);
- int SSL_CTX_set_ssl_version(SSL_CTX *ctx, SSL_METHOD *meth);
- void SSL_CTX_set_timeout(SSL_CTX *ctx, long t);
- long SSL_CTX_set_tmp_dh(SSL_CTX* ctx, DH *dh);
- long SSL_CTX_set_tmp_dh_callback(SSL_CTX *ctx, DH *(*cb)(void));
- long SSL_CTX_set_tmp_rsa(SSL_CTX *ctx, RSA *rsa);
- SSL_CTX_set_tmp_rsa_callback long SSL_CTX_set_tmp_rsa_callback(SSL_CTX *ctx, RSA *(*cb)(SSL *ssl, int export, int keylength));

  Sets the callback which will be called when a temporary private key is required. The export flag will be set if the reason for needing a temp key is that an export ciphersuite is in use, in which case, keylengthwill contain the required keylength in bits. Generate a key of appropriate size and return it.
- SSL_set_tmp_rsa_callback long SSL_set_tmp_rsa_callback(SSL *ssl, RSA *(*cb)(SSL *ssl, int export, int keylength));

  The same as SSL_CTX_set_tmp_rsa_callback, except it operates on an SSL session instead of a context.
- void SSL_CTX_set_verify(SSL_CTX *ctx, int mode, int (*cb);(void));
- int SSL_CTX_use_PrivateKey(SSL_CTX *ctx, EVP_PKEY *pkey);
- int SSL_CTX_use_PrivateKey_ASN1(int type, SSL_CTX *ctx, unsigned char *d, long len);
- int SSL_CTX_use_PrivateKey_file(SSL_CTX *ctx, char *file, int type);
- int SSL_CTX_use_RSAPrivateKey(SSL_CTX *ctx, RSA *rsa);
- int SSL_CTX_use_RSAPrivateKey_ASN1(SSL_CTX *ctx, unsigned char *d, long len);
- int SSL_CTX_use_RSAPrivateKey_file(SSL_CTX *ctx, char *file, int type);
- int SSL_CTX_use_certificate(SSL_CTX *ctx, X509 *x);
- int SSL_CTX_use_certificate_ASN1(SSL_CTX *ctx, int len, unsigned char *d);
- int SSL_CTX_use_certificate_file(SSL_CTX *ctx, char *file, int type);

Dealing with Sessions

The API functions that deal with the SSL/TLS sessions defined in the SSL_SESSIONstructures are described in the following list:

- int SSL_SESSION_cmp(SSL_SESSION *a, SSL_SESSION *b);
- void SSL_SESSION_free(SSL_SESSION *ss);
- char *SSL_SESSION_get_app_data(SSL_SESSION *s);
- char *SSL_SESSION_get_ex_data(SSL_SESSION *s, int idx);
- int SSL_SESSION_get_ex_new_index(long argl, char *argp, int (*new_func);(void), int (*dup_func)(void), void (*free_func)(void));
- long SSL_SESSION_get_time(SSL_SESSION *s);
- long SSL_SESSION_get_timeout(SSL_SESSION *s);
- unsigned long SSL_SESSION_hash(SSL_SESSION *a);
- SSL_SESSION *SSL_SESSION_new(void);
- int SSL_SESSION_print(BIO *bp, SSL_SESSION *x);
- int SSL_SESSION_print_fp(FILE *fp, SSL_SESSION *x);
- void SSL_SESSION_set_app_data(SSL_SESSION *s, char *a);
- int SSL_SESSION_set_ex_data(SSL_SESSION *s, int idx, char *arg);
- long SSL_SESSION_set_time(SSL_SESSION *s, long t);
- long SSL_SESSION_set_timeout(SSL_SESSION *s, long t);

Dealing with Connections

The API functions that deal with the SSL/TLS connection defined in the SSL structure are described in the following list:

- int SSL_accept(SSL *ssl);
- int SSL_add_dir_cert_subjects_to_stack(STACK *stack, const char *dir);
- int SSL_add_file_cert_subjects_to_stack(STACK *stack, const char *file);
- int SSL_add_client_CA(SSL *ssl, X509 *x);
- char *SSL_alert_desc_string(int value);
- char *SSL_alert_desc_string_long(int value);
- char *SSL_alert_type_string(int value);
- char *SSL_alert_type_string_long(int value);
- int SSL_check_private_key(SSL *ssl);
- void SSL_clear(SSL *ssl);
- long SSL_clear_num_renegotiations(SSL *ssl);
- int SSL_connect(SSL *ssl);
- void SSL_copy_session_id(SSL *t, SSL *f);
- long SSL_ctrl(SSL *ssl, int cmd, long larg, char *parg);
- int SSL_do_handshake(SSL *ssl);
- SSL *SSL_dup(SSL *ssl);
- STACK *SSL_dup_CA_list(STACK *sk);

- void SSL_free(SSL *ssl);
- SSL_CTX *SSL_get_SSL_CTX(SSL *ssl);
- char *SSL_get_app_data(SSL *ssl);
- X509 *SSL_get_certificate(SSL *ssl);
- const char *SSL_get_cipher(SSL *ssl);
- int SSL_get_cipher_bits(SSL *ssl, int *alg_bits);
- char *SSL_get_cipher_list(SSL *ssl, int n);
- char *SSL_get_cipher_name(SSL *ssl);
- char *SSL_get_cipher_version(SSL *ssl);
- STACK *SSL_get_ciphers(SSL *ssl);
- STACK *SSL_get_client_CA_list(SSL *ssl);
- SSL_CIPHER *SSL_get_current_cipher(SSL *ssl);
- long SSL_get_default_timeout(SSL *ssl);
- int SSL_get_error(SSL *ssl, int i);
- char *SSL_get_ex_data(SSL *ssl, int idx);
- int SSL_get_ex_data_X509_STORE_CTX_idx(void);
- int SSL_get_ex_new_index(long argl, char *argp, int (*new_func);(void), int (*dup_func)(void), void (*free_func)(void));
- int SSL_get_fd(SSL *ssl);
- void (*SSL_get_info_callback(SSL *ssl);)(void);
- STACK *SSL_get_peer_cert_chain(SSL *ssl);
- X509 *SSL_get_peer_certificate(SSL *ssl);
- EVP_PKEY *SSL_get_privatekey(SSL *ssl);
- int SSL_get_quiet_shutdown(SSL *ssl);
- BIO *SSL_get_rbio(SSL *ssl);
- int SSL_get_read_ahead(SSL *ssl);
- SSL_SESSION *SSL_get_session(SSL *ssl);
- char *SSL_get_shared_ciphers(SSL *ssl, char *buf, int len);
- int SSL_get_shutdown(SSL *ssl);
- SSL_METHOD *SSL_get_ssl_method(SSL *ssl);
- int SSL_get_state(SSL *ssl);
- long SSL_get_time(SSL *ssl);
- long SSL_get_timeout(SSL *ssl);
- int (*SSL_get_verify_callback(SSL *ssl);)(void);
- int SSL_get_verify_mode(SSL *ssl);
- long SSL_get_verify_result(SSL *ssl);

- char *SSL_get_version(SSL *ssl);
- BIO *SSL_get_wbio(SSL *ssl);
- int SSL_in_accept_init(SSL *ssl);
- int SSL_in_before(SSL *ssl);
- int SSL_in_connect_init(SSL *ssl);
- int SSL_in_init(SSL *ssl);
- int SSL_is_init_finished(SSL *ssl);
- STACK *SSL_load_client_CA_file(char *file);
- void SSL_load_error_strings(void);
- SSL *SSL_new(SSL_CTX *ctx);
- long SSL_num_renegotiations(SSL *ssl);
- int SSL_peek(SSL *ssl, void *buf, int num);
- int SSL_pending(SSL *ssl);
- int SSL_read(SSL *ssl, void *buf, int num);
- int SSL_renegotiate(SSL *ssl);
- char *SSL_rstate_string(SSL *ssl);
- char *SSL_rstate_string_long(SSL *ssl);
- long SSL_session_reused(SSL *ssl);
- void SSL_set_accept_state(SSL *ssl);
- void SSL_set_app_data(SSL *ssl, char *arg);
- void SSL_set_bio(SSL *ssl, BIO *rbio, BIO *wbio);
- int SSL_set_cipher_list(SSL *ssl, char *str);
- void SSL_set_client_CA_list(SSL *ssl, STACK *list);
- void SSL_set_connect_state(SSL *ssl);
- int SSL_set_ex_data(SSL *ssl, int idx, char *arg);
- int SSL_set_fd(SSL *ssl, int fd);
- void SSL_set_info_callback(SSL *ssl, void (*cb);(void));
- void SSL_set_options(SSL *ssl, unsigned long op);
- void SSL_set_quiet_shutdown(SSL *ssl, int mode);
- void SSL_set_read_ahead(SSL *ssl, int yes);
- int SSL_set_rfd(SSL *ssl, int fd);
- int SSL_set_session(SSL *ssl, SSL_SESSION *session);
- void SSL_set_shutdown(SSL *ssl, int mode);
- int SSL_set_ssl_method(SSL *ssl, SSL_METHOD *meth);
- void SSL_set_time(SSL *ssl, long t);
- void SSL_set_timeout(SSL *ssl, long t);

- void SSL_set_verify(SSL *ssl, int mode, int (*callback);(void));
- void SSL_set_verify_result(SSL *ssl, long arg);
- int SSL_set_wfd(SSL *ssl, int fd);
- int SSL_shutdown(SSL *ssl);
- int SSL_state(SSL *ssl);
- char *SSL_state_string(SSL *ssl);
- char *SSL_state_string_long(SSL *ssl);
- long SSL_total_renegotiations(SSL *ssl);
- int SSL_use_PrivateKey(SSL *ssl, EVP_PKEY *pkey);
- int SSL_use_PrivateKey_ASN1(int type, SSL *ssl, unsigned char *d, long len);
- int SSL_use_PrivateKey_file(SSL *ssl, char *file, int type);
- int SSL_use_RSAPrivateKey(SSL *ssl, RSA *rsa);
- int SSL_use_RSAPrivateKey_ASN1(SSL *ssl, unsigned char *d, long len);
- int SSL_use_RSAPrivateKey_file(SSL *ssl, char *file, int type);
- int SSL_use_certificate(SSL *ssl, X509 *x);
- int SSL_use_certificate_ASN1(SSL *ssl, int len, unsigned char *d);
- int SSL_use_certificate_file(SSL *ssl, char *file, int type);
- int SSL_version(SSL *ssl);
- int SSL_want(SSL *ssl);
- int SSL_want_nothing(SSL *ssl);
- int SSL_want_read(SSL *ssl);
- int SSL_want_write(SSL *ssl);
- int SSL_want_x509_lookup(s);
- int SSL_write(SSL *ssl, const void *buf, int num);

HISTORY

The SSL document appeared in OpenSSL 0.9.2.

SEE ALSO

Commands: *openssl*

Functions: *crypto*, *SSL_accept*, *SSL_clear*, *SSL_connect*, *SSL_CIPHER_get_name*, *SSL_CTX_add_extra_chain_cert*, *SSL_CTX_add_session*, *SSL_CTX_flush_sessions*, *SSL_CTX_get_ex_new_index*, *SSL_CTX_get_verify_mode*, *SSL_CTX_load_verify_locations*, *SSL_CTX_new*, *SSL_CTX_sess_number*, *SSL_CTX_sess_set_cache_size*, *SSL_CTX_sess_set_get_cb*, *SSL_CTX_sessions*, *SSL_CTX_set_client_CA_list*, *SSL_CTX_set_default_passwd_cb*, *SSL_CTX_set_mode*, *SSL_CTX_set_options*, *SSL_CTX_set_session_cache_mode*, *SSL_CTX_set_session_id_context*, *SSL_CTX_set_ssl_version*, *SSL_CTX_set_timeout*, *SSL_CTX_set_verify*, *SSL_CTX_use_certificate*, *SSL_get_ciphers*, *SSL_get_client_CA_list*, *SSL_get_error*, *SSL_get_ex_data_X509_STORE_CTX_idx*, *SSL_get_ex_new_index*, *SSL_get_fd*, *SSL_get_peer_cert_chain*, *SSL_get_rbio*, *SSL_get_session*, *SSL_get_verify_result*,

SSL_get_version, *SSL_library_init*, *SSL_load_client_CA_file*, *SSL_new*, *SSL_read*, *SSL_set_bio*, *SSL_set_connect_state*, *SSL_set_fd*, *SSL_pending*, *SSL_set_session*, *SSL_set_shutdown*, *SSL_shutdown*, *SSL_write*, *SSL_SESSION_free*, *SSL_SESSION_get_ex_new_index*, *SSL_SESSION_get_time*, *d2i_SSL_SESSION*

SSL_accept

NAME

SSL_accept – Wait for a TLS/SSL client to initiate a TLS/SSL handshake

SYNOPSIS

```
#include <openssl/ssl.h>
int SSL_accept(
        SSL *ssl
);
```

DESCRIPTION

The `SSL_accept()` function waits for a TLS/SSL client to initiate the TLS/SSL handshake. The communication channel must already have been set and assigned to the `ssl` by setting an underlying BIO.

NOTES

The behavior of the `SSL_accept()` function depends on the underlying BIO.

If the underlying BIO is `blocking`, the `SSL_accept()` function will only return once the handshake has been finished or an error occurred, except for Server Gated Cryptography (SGC). For SGC, the `SSL_accept()` function might return with -1, but the `SSL_get_error()` function will yield `SSL_ERROR_WANT_READ/WRITE` and `SSL_accept()` should be called again.

If the underlying BIO is `non-blocking`, the `SSL_accept()` function will also return when the underlying BIO could not satisfy the needs of the `SSL_accept()` function to continue the handshake. In this case a call to the `SSL_get_error()` function with the return value of `SSL_accept()` will yield `SSL_ERROR_WANT_READ` or `SSL_ERROR_WANT_WRITE`. The calling process then must repeat the call after taking appropriate action to satisfy the needs of `SSL_accept()`. The action depends on the underlying BIO. When using a non-blocking socket, nothing is to be done, but the `select()` function can be used to check for the required condition. When using a buffering BIO, like a BIO pair, data must be written into or retrieved out of the BIO before being able to continue.

When using a generic method (see *SSL_CTX_new*), it is necessary to call the `SSL_set_accept_state()` function before calling `SSL_accept()` to explicitly switch the `ssl` to server mode.

RETURN VALUES

The following return values can occur:

1

The TLS/SSL handshake was successfully completed, a TLS/SSL connection has been established.

0

The TLS/SSL handshake was not successful but was shut down controlled and by the specifications of the TLS/SSL protocol. Call `SSL_get_error()` with the return value `ret` to find the reason.

<0

The TLS/SSL handshake was not successful because a fatal error occurred either at the protocol level or a connection failure occurred. The shutdown was not clean. It can also occur if action is needed to continue the operation for non-blocking BIOs. Call the `SSL_get_error()` function with the return value `ret` to find the reason.

SEE ALSO

Functions: *SSL_get_error*, *SSL_connect*, *SSL_shutdown*, *ssl*, *bio*, *SSL_set_connect_state*, *SSL_CTX_new*

SSL_alert_desc_string

NAME

SSL_alert_desc_string, SSL_alert_desc_string_long – Get description of an SSL alert

SYNOPSIS

```
#include <openssl/ssl.h>
char *SSL_alert_desc_string(
        int value
);
char *SSL_alert_desc_string_long(
        int value
);
```

DESCRIPTION

The `SSL_alert_desc_string()` returns two-letter strings indicating SSL alerts.

The `SSL_alert_desc_string_long()` returns message strings about SSL alerts.

RETURN VALUES

CN | "close notify"

Indicates that the "close notify" SSL alert was received.

UM | "unexpected_message"

Indicates that the "unexpected_message" SSL alert was received.

BM | "bad record mac"

Indicates that the "bad record mac" SSL alert was received.

DF | "decompression failure"

Indicates that the "decompression failure" SSL alert was received.

HF | "handshake failure"

Indicates that the "handshake failure" SSL alert was received.

NC | "no certificate"

Indicates that the "no certificate" SSL alert was received.

BC | "bad certificate"

Indicates that the "bad certificate" SSL alert was received.

UC | "unsupported certificate"

Indicates that the "unsupported certificate" SSL alert was received.

CR | "certificate revoked"

Indicates that the "certificate revoked" SSL alert was received.

CE | "certificate expired"

Indicates that the "certificate expired" SSL alert was received.

CU | "certificate unknown"

Indicates that the "certificate unknown" SSL alert was received.

IP | "illegal parameter"

Indicates that the "illegal parameter" SSL alert was received.

BC | "unknown"

Indicates that the "unknown" SSL alert was received.

SSL_alert_type_string

NAME

SSL_alert_type_string, SSL_alert_type_string_long, SSL_alert_desc_string, SSL_alert_desc_string_long – Get textual description of alert information

SYNOPSIS

```
#include <openssl/ssl.h>
char *SSL_alert_type_string(
        int value
);
char *SSL_alert_type_string_long(
        int value
);
char *SSL_alert_desc_string(
        int value
);
char *SSL_alert_desc_string_long(
        int value
);
```

DESCRIPTION

The `SSL_alert_type_string()` function returns a one letter string indicating the type of the alert specified by value.

The `SSL_alert_type_string_long()` function returns a string indicating the type of the alert specified by value.

The `SSL_alert_desc_string()` function returns a two letter string as a short form describing the reason of the alert specified by value.

The `SSL_alert_desc_string_long()` function returns a string describing the reason of the alert specified by value.

NOTES

When one side of an SSL/TLS communication wants to inform the peer about a special situation, it sends an alert. The alert is sent as a special message and does not influence the normal data stream, unless its contents result in the communication being canceled.

A warning alert is sent when a non-fatal error condition occurs. The ``close notify'' alert is sent as a warning alert. Other examples of non-fatal errors are certificate errors, such as -certificate expired'' and -unsupported certificate,- for which a warning alert might be sent. (The sending party might decide to send a fatal error.) The receiving side, at its discretion, can cancel the connection after receiving a warning alert.

Several alert messages must be sent as fatal alert messages as specified by the TLS RFC. A fatal alert always leads to a connection abort.

RETURN VALUES

The `SSL_alert_type_string()` or `SSL_alert_type_string_long()` functions return a one letter string indicating the type of the alert specified by value:

W

Warning

F

Fatal

U

Unknown

This indicates that no support is available for this alert type. Probably value does not contain a correct alert message.

The following strings can occur for the `SSL_alert_desc_string()` or `SSL_alert_desc_string_long()` functions:

CN

close notify

The connection will be closed. This is a warning alert.

UM

unexpected message

An inappropriate message was received. This alert is always fatal and should never be observed in communication between proper implementations.

BM

bad record mac

This alert is returned if a record is received with an incorrect MAC. This message is always fatal.

DF

decompression failure

The decompression function received improper input (e.g. data that would expand to excessive length). This message is always fatal.

HF

handshake failure

Reception of a handshake_failure alert message indicates that the sender was unable to negotiate an acceptable set of security parameters given the options available. This is a fatal error.

NC

no certificate

A client, that was asked to send a certificate, does not send a certificate (SSLv3 only).

BC

bad certificate

A certificate was corrupt, contained signatures that did not verify correctly, etc.

UC

unsupported certificate

A certificate was of an unsupported type.

CR

certificate revoked

A certificate was revoked by its signer.

CE

certificate expired

A certificate has expired or is not currently valid.

CU

certificate unknown

Some other (unspecified) issue arose in processing the certificate, rendering it unacceptable.

IP

illegal parameter

A field in the handshake was out of range or inconsistent with other fields. This is always fatal.

DC

decryption failed

A TLSCiphertext decrypted in an invalid way: either it wasn't an even multiple of the block length or its padding values, when checked, weren't correct. This message is always fatal.

RO

record overflow

A TLSCiphertext record was received which had a length more than 2^14+2048 bytes, or a record decrypted to a TLSCompressed record with more than 2^14+1024 bytes. This message is always fatal.

CA

unknown CA

A valid certificate chain or partial chain was received, but the certificate was not accepted because the CA certificate could not be located or couldn't be matched with a known, trusted CA. This message is always fatal.

AD

access denied

A valid certificate was received, but when access control was applied, the sender decided not to proceed with negotiation. This message is always fatal.

DE

decode error

A message could not be decoded because some field was out of the specified range or the length of the message was incorrect. This message is always fatal.

CY

decrypt error

A handshake cryptographic operation failed, including being unable to correctly verify a signature, decrypt a key exchange, or validate a finished message.

ER

export restriction

A negotiation not in compliance with export restrictions was detected; for example, attempting to transfer a 1024 bit ephemeral RSA key for the RSA_EXPORT handshake method. This message is always fatal.

PV

protocol version

The protocol version the client has attempted to negotiate is recognized, but not supported. (For example, old protocol versions might be avoided for security reasons). This message is always fatal.

IS

insufficient security

Returned instead of `handshake_failure` when a negotiation fails, specifically because the server requires ciphers more secure than those supported by the client. This message is always fatal.

IE

internal error

An internal error unrelated to the peer or the correctness of the protocol makes it impossible to continue (such as a memory allocation failure). This message is always fatal.

US

user cancelled

This handshake is being canceled for some reason unrelated to a protocol failure. If the user cancels an operation after the handshake is complete, just closing the connection by sending a close_notify is more appropriate. This alert should be followed by a close_notify. This message is generally a warning.

NR

no renegotiation

Sent by the client in response to a hello request or by the server in response to a client hello after initial handshaking. Either of these would normally lead to renegotiation; when that is not appropriate, the recipient should respond with this alert; at that point, the original requester can decide whether to proceed with the connection. One case where this would be appropriate would be where a server has spawned a process to satisfy a request; the process might receive security parameters (key length, authentication, etc.) at startup and it might be difficult to communicate changes to these parameters after that point. This message is always a warning.

UK

unknown

This indicates that no description is available for this alert type. Probably value does not contain a correct alert message.

SEE ALSO

Functions: *ssl*

SSL_callback_ctrl

NAME

SSL_callback_ctrl – Perform an operation (get or set information in SSL) for the SSL structure

SYNOPSIS

```
#include <openssl/ssl.h>
long SSL_ctrl(
        int  cmd)
        (long larg)
        (char *parg)
        (SSL * s
);
```

DESCRIPTION

The `SSL_ctrl()` function performs an operation (get or set information in SSL) for the SSL structure. The second argument `cmd` accepts the macros in the following table:

Table 1 Macros for cmd argument in SSL_callback_ctrl

	SSLv2	**SSLv3**	**TLSv1**
SSL_CTRL_GET_READ_AHEAD	YES	YES	YES
SSL_CTRL_SET_READ_AHEAD	YES	YES	YES
SSL_CTRL_OPTIONS	YES	YES	YES
SSL_CTRL_MODE	YES	YES	YES
SSL_CTRL_GET_SESSION_REUSED	YES	YES	NO
SSL_CTRL_GET_CLIENT_CERT_REQUEST	NO	YES	NO
SSL_CTRL_GET_NUM_RENEGOTIATIONS	NO	YES	NO
SSL_CTRL_CLEAR_NUM_RENEGOTIATIONS	NO	YES	NO
SSL_CTRL_GET_TOTAL_RENEGOTIATIONS	NO	YES	NO
SSL_CTRL_GET_FLAGS	NO	YES	NO
SSL_CTRL_NEED_TMP_RSA	NO	YES(#ifndef NO_RSA)	NO
SSL_CTRL_SET_TMP_RSA	NO	YES(#ifndef NO_RSA)	NO
SSL_CTRL_SET_TMP_RSA_CB	NO	YES(#ifndef NO_RSA)	NO
SSL_CTRL_SET_TMP_DH	NO	YES(#ifndef NO_DH)	NO

Table 1 **Macros for cmd argument in SSL_callback_ctrl (Continued)**

	SSLv2	SSLv3	TLSv1
SSL_CTRL_SET_TMP_DH_CB	NO	YES(#ifndef NO_DH)	NO

RETURN VALUES

The `SSL_ctrl()` function returns a long. The return value depends on the type of command `cmd` passed to this API.

SEE ALSO

Functions: *SSL_CTX_ctrl*, *SSL_callback_ctrl*, *SSL_CTX_callback_ctrl*

SSL_check_private_key

NAME

SSL_check_private_key – Checks the private key against the public key of the certificate in the SSL structure

SYNOPSIS

```
#include <openssl/ssl.h>
int SSL_check_private_key(
       SSL *ssl
);
```

DESCRIPTION

The `SSL_check_private_key()` function checks if the private key matches against the public key of the certificate loaded in the SSL structure.

RETURN VALUES

The following return values can occur:

1

The private key matches against the public key in the SSL structure.

0

The verification of the private key failed.

NOTES

This API does not implement the functionality of checking DH keys.

SSL_CIPHER_get_name

NAME

SSL_CIPHER_get_name, SSL_CIPHER_get_bits, SSL_CIPHER_get_version, SSL_CIPHER_description – Get SSL_CIPHER properties

SYNOPSIS

```
#include <openssl/ssl.h>
const char *SSL_CIPHER_get_name(
        SSL_CIPHER *cipher
);
int SSL_CIPHER_get_bits(
        SSL_CIPHER *cipher, int *alg_bits
);
char *SSL_CIPHER_get_version(
        SSL_CIPHER *cipher
);
char *SSL_CIPHER_description(
        SSL_CIPHER *cipher, char *buf, int size
);
```

DESCRIPTION

The `SSL_CIPHER_get_name()` function returns a pointer to the name of `cipher`. If the argument is the NULL pointer, a pointer to the constant value NONE is returned.

The `SSL_CIPHER_get_bits()` function returns the number of secret bits used for `cipher`. If `alg_bits` is not NULL, it contains the number of bits processed by the chosen algorithm. If `cipher` is NULL, 0 is returned.

The `SSL_CIPHER_get_version()` function returns the protocol version for `cipher`, currently SSLv2, SSLv3, or TLSv1. If `cipher` is NULL, NONE is returned.

The `SSL_CIPHER_description()` function returns a textual description of the cipher used into the buffer (`buf`) of length (`len`) provided. The `len` must be at least 128 bytes, otherwise a pointer to the the string "Buffer too small" is returned. If `buf` is NULL, a buffer of 128 bytes is allocated using the `OPENSSL_malloc()` function. If the allocation fails, a pointer to the string "OPENSSL_malloc Error" is returned.

NOTES

The number of bits processed can be different from the secret bits. An export cipher, such as EXP-RC4-MD5, has 40 secret bits. The algorithm uses the full 128 bits (which would be returned for `alg_bits`), of which 88 bits are fixed. The search space is 40 bits.

The string returned by the `SSL_CIPHER_description()` function in case of success consists of cleartext information separated by one or more blanks in the following sequence:

<ciphername>

Textual representation of the cipher name.

<protocol version>

Protocol version: SSLv2, SSLv3. The TLSv1 ciphers are flagged with SSLv3.

Kx=<key exchange>

Key exchange method: RSA (for export ciphers as RSA(512) or RSA(1024)), DH (for export ciphers as DH(512) or DH(1024)), DH/RSA, DH/DSS, Fortezza.

Au=<authentication>

Authentication method: RSA, DSS, DH, None. None is the representation of anonymous ciphers.

Enc=<symmetric encryption method>

Encryption method with number of secret bits: DES(40), DES(56), 3DES(168), RC4(40), RC4(56), RC4(64), RC4(128), RC2(40), RC2(56), RC2(128), IDEA(128), Fortezza, None.

Mac=<message authentication code>

Message digest: MD5, SHA1.

<export flag>

If the cipher is flagged exportable with respect to old US crypto regulations, the word "export" is printed.

RESTRICTIONS

If the `SSL_CIPHER_description()` function is called with `cipher` being NULL, the library crashes.

If the `SSL_CIPHER_description()` function cannot handle a built-in cipher, the description of the cipher property is `unknown`. This case should not occur.

RETURN VALUES

See Description

EXAMPLES

The following examples show output for the `SSL_CIPHER_description()` function:

```
EDH-RSA-DES-CBC3-SHA     SSLv3 Kx=DH       Au=RSA  Enc=3DES(168) Mac=SHA1
EDH-DSS-DES-CBC3-SHA     SSLv3 Kx=DH       Au=DSS  Enc=3DES(168) Mac=SHA1
RC4-MD5                  SSLv3 Kx=RSA      Au=RSA  Enc=RC4(128)  Mac=MD5
EXP-RC4-MD5              SSLv3 Kx=RSA(512) Au=RSA  Enc=RC4(40)   Mac=MD5  export
```

SEE ALSO

Commands: *ciphers*

Functions: *ssl*, *SSL_get_current_cipher*, *SSL_get_ciphers*

SSL_clear

NAME

SSL_clear – Reset SSL object to allow another connection

SYNOPSIS

```
#include <openssl/ssl.h>
int SSL_clear(
        SSL *ssl
);
```

DESCRIPTION

Reset `ssl` to allow another connection. All settings (method, ciphers, BIOs) are kept.

NOTES

The `SSL_clear()` function is used to prepare an SSL object for a new connection. While all settings are kept, a side effect is the handling of the current SSL session. If a session is still open, it is considered bad and will be removed from the session cache, as required by RFC2246. A session is considered open, if the `SSL_shutdown()` function was not called for the connection or at least the `SSL_set_shutdown()` function was used to set the SSL_SENT_SHUTDOWN state.

RETURN VALUES

The following return values can occur:

0

The `SSL_clear()` function operation could not be performed. Check the error stack to find out the reason.

1

The `SSL_clear()` function operation was successful.

SEE ALSO

Functions: *SSL_new*, *SSL_free*, *SSL_shutdown*, *SSL_set_shutdown*, *SSL_CTX_set_options*, *ssl*

SSL_COMP_add_compression_method

NAME

SSL_COMP_add_compression_method – Handle SSL/TLS integrated compression methods

SYNOPSIS

```
#include <openssl/ssl.h>
int SSL_COMP_add_compression_method(
        int id)
        (COMP_METHOD  *cm
);
```

DESCRIPTION

The `SSL_COMP_add_compression_method()` function adds the compression method `cm` with the identifier `id` to the list of available compression methods. This list is globally maintained for all SSL operations within this application. It cannot be set for specific SSL_CTX or SSL objects.

NOTES

The TLS standard (or SSLv3) allows the integration of compression methods into the communication. However, the TLS RFC does not specify compression methods or their corresponding identifiers, so there is currently no compatible way to integrate compression with unknown peers. We do not recommended integrating compression into applications. Applications for non-public use may agree on certain compression methods. Using different compression methods with the same identifier will lead to connection failure.

An OpenSSL client speaking a protocol that allows compression (SSLv3, TLSv1) will unconditionally send the list of all compression methods enabled with `SSL_COMP_add_compression_method()` to the server during the handshake. Unlike the mechanisms to set a cipher list, there is no method available to restrict the list of compression method on a per connection basis.

An OpenSSL server will match the identifiers listed by a client against its own compression methods and will unconditionally activate compression when a matching identifier is found. There is no way to restrict the list of compression methods supported on a per connection basis.

The OpenSSL library has the compression methods `COMP_rle()` and (when especially enabled during compilation) `COMP_zlib()` available.

Once the identities of the compression methods for the TLS protocol have been standardized, the compression API will most likely be changed. We do not recommended using it in its current state.

RETURN VALUES

The following return values can occur:

0

The operation failed. Check the error queue to find out the reason.

1

The operation succeeded.

SSL_connect

NAME

SSL_connect – Initiate the TLS/SSL handshake with an TLS/SSL server

SYNOPSIS

```
#include <openssl/ssl.h>
int SSL_connect(
        SSL *ssl
);
```

DESCRIPTION

The `SSL_connect()` function initiates the TLS/SSL handshake with a server. The communication channel must already have been set and assigned to the `ssl` by setting an underlying BIO.

NOTES

The behavior of the `SSL_connect()` function depends on the underlying BIO.

If the underlying BIO is `blocking`, the `SSL_connect()` function will only return once the handshake has been finished or an error occurred.

If the underlying BIO is `non-blocking`, the `SSL_connect()` function will also return when the underlying BIO could not satisfy the needs of `SSL_connect()` to continue the handshake. In this case, a call to `SSL_get_error()` with the return value of `SSL_connect()` will yield SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process then must repeat the call after taking appropriate action to satisfy the needs of `SSL_connect()`. The action depends on the underlying BIO. When using a non-blocking socket, nothing is to be done, but the `select()` function can be used to check for the required condition. When using a buffering BIO, like a BIO pair, data must be written into or retrieved out of the BIO before being able to continue.

When using a generic method (see *SSL_CTX_new*), it is necessary to call `SSL_set_connect_state()` before calling `SSL_connect()` to explicitly switch the `ssl` to client mode.

RETURN VALUES

The following return values can occur:

1

The TLS/SSL handshake was successfully completed. A TLS/SSL connection was established.

0

The TLS/SSL handshake was not successful but was shut down controlled and by the specifications of the TLS/SSL protocol. Call the `SSL_get_error()` function with the return value `ret` to find out the reason.

<0

The TLS/SSL handshake was not successful, because a fatal error occurred either at the protocol level or a connection failure occurred. The shutdown was not clean. It can also occur if action is needed to continue the operation for non-blocking BIOs. Call the `SSL_get_error()` function with the return value `ret` to find out the reason.

SEE ALSO

Functions: *SSL_get_error*, *SSL_accept*, *SSL_shutdown*, *ssl*, *bio*, *SSL_set_connect_state*, *SSL_CTX_new*

SSL_copy_session_id

NAME

SSL_copy_session_id – Copies the session-id from one SSL structure to another

SYNOPSIS

```
#include <openssl/ssl.h>
void SSL_copy_session_id(
        SSL *t)
       (SSL *f
);
```

DESCRIPTION

The `SSL_copy_session_id()` function copies an SSL session-id from SSL structure `f` and to SSL structure `t`.

SEE ALSO

Functions: *BIO_ssl_copy_session_id*

SSL_ctrl

NAME

SSL_ctrl – Performs an operation (get or set information in SSL) for the SSL structure

SYNOPSIS

```
#include <openssl/ssl.h>
long SSL_ctrl(
        SSL * s)
        (int cmd)
        (long larg)
        (char * parg
);
```

DESCRIPTION

The `SSL_ctrl()` function performs an operation (get or set information in SSL) for the SSL structure. The second argument `cmd` accepts the macros in the following table:

Table 2 Macros for cmd argument in SSL_ctrl

	SSLv2	SSLv3	TLSv1
SSL_CTRL_GET_READ_AHEAD	YES	YES	YES
SSL_CTRL_SET_READ_AHEAD	YES	YES	YES
SSL_CTRL_OPTIONS	YES	YES	YES
SSL_CTRL_MODE	YES	YES	YES
SSL_CTRL_GET_SESSION_REUSED	YES	YES	NO
SSL_CTRL_GET_CLIENT_CERT_REQUEST	NO	YES	NO
SSL_CTRL_GET_NUM_RENEGOTIATIONS	NO	YES	NO
SSL_CTRL_CLEAR_NUM_RENEGOTIATIONS	NO	YES	NO
SSL_CTRL_GET_TOTAL_RENEGOTIATIONS	NO	YES	NO
SSL_CTRL_GET_FLAGS	NO	YES	NO
SSL_CTRL_NEED_TMP_RSA	NO	YES(#ifndef NO_RSA)	NO
SSL_CTRL_SET_TMP_RSA	NO	YES(#ifndef NO_RSA)	NO
SSL_CTRL_SET_TMP_RSA_CB	NO	YES(#ifndef NO_RSA)	NO
SSL_CTRL_SET_TMP_DH	NO	YES(#ifndef NO_DH)	NO
SSL_CTRL_SET_TMP_DH_CB	NO	YES(#ifndef NO_DH)	NO

RETURN VALUES

The `SSL_ctrl()` function returns a long. The return value depends on the type of command `cmd` passed to this API.

SEE ALSO

Functions: *SSL_CTX_ctrl*, *SSL_callback_ctrl*, *SSL_CTX_callback_ctrl*

SSL_CTX_add_extra_chain_cert

NAME

SSL_CTX_add_extra_chain_cert – Add certificate to chain

SYNOPSIS

```
#include <openssl/ssl.h>
long SSL_CTX_add_extra_chain_cert(
        SSL_CTX ctx, X509 *x509
);
```

DESCRIPTION

The `SSL_CTX_add_extra_chain_cert()` function adds the certificate `x509` to the certificate chain presented together with the certificate. Several certificates can be added one after the other.

NOTES

When constructing the certificate chain, the chain will be formed from these certificates explicitly specified. If no chain is specified, the library will try to complete the chain from the available CA certificates in the trusted CA storage. See `SSL_CTX_load_verify_locations()`.

RETURN VALUES

The `SSL_CTX_add_extra_chain_cert()` function returns 1 on success. Check the error stack to determine the reason for failure.

SEE ALSO

Functions: *ssl*, *SSL_CTX_use_certificate*, *SSL_CTX_load_verify_locations*

SSL_CTX_add_session

NAME

SSL_CTX_add_session, SSL_add_session, SSL_CTX_remove_session, SSL_remove_session – Manipulate session cache

SYNOPSIS

```
#include <openssl/ssl.h>
int SSL_CTX_add_session(
        SSL_CTX *ctx, SSL_SESSION *c
);
int SSL_add_session(
       SSL_CTX *ctx, SSL_SESSION *c
);
int SSL_CTX_remove_session(
        SSL_CTX *ctx, SSL_SESSION *c
);
int SSL_remove_session(
       SSL_CTX *ctx, SSL_SESSION *c
);
```

DESCRIPTION

The `SSL_CTX_add_session()` function adds the session `c` to the context `ctx`. The reference count for session `c` is incremented by 1. If a session with the same session id already exists, the old session is removed by calling `SSL_SESSION_free()`.

The `SSL_CTX_remove_session()` function removes the session `c` from the context `ctx`. The `SSL_SESSION_free()` function is called once for `c`.

The `SSL_add_session()` and the `SSL_remove_session()` functions are synonyms for their `SSL_CTX_*()` function counterparts.

NOTES

When adding a new session to the internal session cache, it is examined whether a session with the same session id already exists. In this case it is assumed that both sessions are identical. If the same session is stored in a different `SSL_SESSION` object, the old session is removed and replaced by the new session. If the session is identical (the `SSL_SESSION` object is identical), the `SSL_CTX_add_session()` function is a no-op, and the return value is 0.

RETURN VALUES

The following values are returned by all functions:

0

The operation failed. In the case of the add operation, it tried to add the same session twice. In the case of the remove operation, the session was not found in the cache.

1

The operation succeeded.

SEE ALSO

Functions: *ssl*, *SSL_CTX_set_session_cache_mode*, *SSL_SESSION_free*

SSL_CTX_ctrl

NAME

SSL_CTX_ctrl, SSL_CTX_callback_ctrl, SSL_ctrl, SSL_callback_ctrl – Internal handling functions for SSL_CTX and SSL objects

SYNOPSIS

```
#include <openssl/ssl.h>
long SSL_CTX_ctrl(
       SSL_CTX  *ctx)
       (int cmd)
       (long larg)
       (char  *parg
);
long SSL_CTX_callback_ctrl(
        SSL *)
       (int cmd)
       (void (*fp)()
);
long SSL_ctrl(
       SSL_CTX  *ctx)
       (int cmd)
       (long larg)
       (char  *parg
);
long SSL_callback_ctrl(
       SSL  *)
       (int cmd)
       (void (*fp)()
);
```

DESCRIPTION

The `SSL_*_ctrl()` family of functions is used to manipulate settings of the SSL_CTX and SSL objects. Depending on the *cmd* parameter, the arguments *larg*, *parg*, or *fp* are evaluated. These functions should never be called directly. All functionalities needed are made available via other functions or macros.

RETURN VALUES

The return values of the `SSL_CTX_ctrl()` functions depend on the type of command supplied via the *cmd* parameter.

SSL_CTX_flush_sessions

NAME

SSL_CTX_flush_sessions, SSL_flush_sessions – Remove expired sessions

SYNOPSIS

```
#include <openssl/ssl.h>
void SSL_CTX_flush_sessions(
        SSL_CTX *ctx, long tm
);
void SSL_flush_sessions(
        SSL_CTX *ctx, long tm
);
```

DESCRIPTION

The `SSL_CTX_flush_sessions()` function causes a run through the session cache of `ctx` to remove sessions expired at time (`tm`).

The `SSL_flush_sessions()` function is a synonym for the `SSL_CTX_flush_sessions()` function.

NOTES

If enabled, the internal session cache will collect all sessions established up to the specified maximum number. (See `SSL_CTX_sess_set_cache_size()`). As sessions will not be reused once they are expired, they should be removed from the cache to save resources. This can be done automatically whenever 255 new sessions are established (see `SSL_CTX_set_session_cache_mode()`) or manually by calling the `SSL_CTX_flush_sessions()` function.

The parameter `tm` specifies the time which should be used for the expiration test. In most cases the actual time given by *time* will be used.

The `SSL_CTX_flush_sessions()` function will only check sessions stored in the internal cache. When a session is found and removed, the `remove_session_cb` is called to synchronize with the external cache. (See `SSL_CTX_sess_set_get_cb()`.)

SEE ALSO

Functions: *ssl*, *SSL_CTX_set_session_cache_mode*, *SSL_CTX_set_timeout*, *SSL_CTX_sess_set_get_cb*

SSL_CTX_free

NAME

SSL_CTX_free – Free an allocated SSL_CTX object

SYNOPSIS

```
#include <openssl/ssl.h>
void SSL_CTX_free(
        SSL_CTX *ctx
);
```

DESCRIPTION

The `SSL_CTX_free()` function decrements the reference count of `ctx`, and removes the `SSL_CTX` object pointed to by `ctx` and frees up the allocated memory if the the reference count has reached 0.

If applicable, it also calls the freeing procedures for indirectly affected items: the session cache, the list of ciphers, the list of Client CAs, the certificates and keys.

RETURN VALUES

The `SSL_CTX_free()` function does not provide diagnostic information.

SEE ALSO

Functions: *SSL_CTX_new*, *ssl*

SSL_CTX_get_cert_store

NAME

SSL_CTX_get_cert_store – Get the X509_STORE structure in the SSL_CTX structure

SYNOPSIS

```
#include <openssl/ssl.h>
X509_STORE *SSL_CTX_get_cert_store(
       SSL_CTX *ctx
);
```

DESCRIPTION

The `SSL_CTX_get_cert_store()` function gets the X509_STORE structure in the SSL_CTX structure. An X509_STORE structure holds information for certificate verification including cache of trusted certificate, external lookup methods and a pointer to a certificate verification function.

SEE ALSO

Functions: *SSL_CTX_set_cert_store*

SSL_CTX_get_ex_new_index

NAME

SSL_CTX_get_ex_new_index, SSL_CTX_set_ex_data, SSL_CTX_get_ex_data – Internal application specific data functions

SYNOPSIS

```
#include <openssl/ssl.h>
int SSL_CTX_get_ex_new_index(
        long argl, void *argp)
        (CRYPTO_EX_new *new_func)
        (CRYPTO_EX_dup *dup_func)
        (CRYPTO_EX_free *free_func
);
int SSL_CTX_set_ex_data(
        SSL_CTX *ctx, int idx, void *arg
);
void *SSL_CTX_get_ex_data(
        SSL_CTX *ctx, int idx
);
typedef int new_func(
       void *parent, void *ptr, CRYPTO_EX_DATA *ad, int idx, long argl, void *argp
);
typedef void free_func(
       void *parent, void *ptr, CRYPTO_EX_DATA *ad, int idx, long argl, void *argp
);
typedef int dup_func(
       CRYPTO_EX_DATA *to, CRYPTO_EX_DATA *from, void *from_d, int idx, long argl, void
       *argp
);
```

DESCRIPTION

Several OpenSSL structures can have application specific data attached to them. These functions are used internally by OpenSSL to manipulate application specific data attached to a specific structure.

The `SSL_CTX_get_ex_new_index()` function is used to register a new index for application specific data. The `SSL_CTX_set_ex_data()` function is used to store application data at `arg` for `idx` into the `ctx` object. The `SSL_CTX_get_ex_data()` function is used to retrieve the information for `idx` from `ctx`.

See `RSA_get_ex_new_index()` for a description of the functionality of `*_get_ex_new_index()`. The `*_get_ex_data()` and `*_set_ex_data()` functionality is described in `CRYPTO_set_ex_data()`.

SEE ALSO

Functions: *ssl*, *RSA_get_ex_new_index*, *CRYPTO_set_ex_data*

SSL_CTX_get_quiet_shutdown

NAME

SSL_CTX_get_quiet_shutdown – Get the value of the quiet-shutdown flag in the SSL_CTX data structure

SYNOPSIS

```
#include <openssl/ssl.h>
int SSL_CTX_get_quiet_shutdown(
        SSL *ssl
);
```

DESCRIPTION

The `SSL_CTX_get_quiet_shutdown()` function returns a mode of the quiet shutdown flag in the SSL_CTX structure.

RETURN VALUES

0

Indicates that the quiet-shutdown flag of the SSL_CTX structure is turned off.

1

Indicates that the quiet-shutdown flag of the SSL_CTX structure is turned on.

SEE ALSO

Functions: *SSL_CTX_set_quiet_shutdown*, *SSL_set_quiet_shutdown*, *SSL_get_quiet_shutdown*

SSL_CTX_get_verify_mode

NAME

SSL_CTX_get_verify_mode, SSL_get_verify_mode, SSL_CTX_get_verify_depth, SSL_get_verify_depth, SSL_get_verify_callback, SSL_CTX_get_verify_callback – Get currently set verification parameters

SYNOPSIS

```
#include <openssl/ssl.h>
int SSL_CTX_get_verify_mode(
        SSL_CTX *ctx
);
int SSL_get_verify_mode(
        SSL *ssl
);
int SSL_CTX_get_verify_depth(
        SSL_CTX *ctx
);
int SSL_get_verify_depth(
        SSL *ssl
);
int *SSL_CTX_get_verify_callback(
        SSL_CTX *ctx)
       (int X509_STORE_CTX *
);
int *SSL_get_verify_callback(
        SSL *ssl)
       (int, X509_STORE_CTX *
);
```

DESCRIPTION

The `SSL_CTX_get_verify_mode()` function returns the verification mode currently set in `ctx`.

The `SSL_get_verify_mode()` function returns the verification mode currently set in `ssl`.

The `SSL_CTX_get_verify_depth()` function returns the verification depth limit currently set in `ctx`. If no limit has been explicitly set, -1 is returned and the default value will be used.

The `SSL_get_verify_depth()` function returns the verification depth limit currently set in `ssl`. If no limit has been explicitly set, -1 is returned and the default value will be used.

The `SSL_CTX_get_verify_callback()` function returns a function pointer to the verification callback currently set in `ctx`. If no callback was explicitly set, the NULL pointer is returned and the default callback will be used.

The `SSL_get_verify_callback()` function returns a function pointer to the verification callback currently set in `ssl`. If no callback was explicitly set, the NULL pointer is returned and the default callback will be used.

RETURN VALUES

See Description.

SEE ALSO

Functions: *ssl*, *SSL_CTX_set_verify*

SSL_CTX_load_verify_locations

NAME

SSL_CTX_load_verify_locations – Set default locations for trusted CA certificates

SYNOPSIS

```
#include <openssl/ssl.h>

int SSL_CTX_load_verify_locations(

       SSL_CTX *ctx, const char *CAfile, const char *CApath

);
```

DESCRIPTION

The `SSL_CTX_load_verify_locations()` function specifies the locations for `ctx`, at which CA certificates for verification purposes are located. The certificates available via `CAfile` and `CApath` are trusted.

NOTES

If `CAfile` is not NULL, it points to a file of CA certificates in PEM format. The file can contain several CA certificates identified by the following sequences:

```
-----BEGIN CERTIFICATE-----
... (CA certificate in base64 encoding) ...
-----END CERTIFICATE-----
```

Text is allowed before, between, and after the certificates. It can be used, for example, to describe the certificates.

The `CAfile` is processed on execution of the `SSL_CTX_load_verify_locations()` function.

If `CApath` is not NULL, it points to a directory containing CA certificates in PEM format. The files each contain one CA certificate. The files are looked up by the CA subject name hash value, which must be available. If more than one CA certificate with the same name hash value exist, the extension must be different (e.g. 9d66eef0.0, 9d66eef0.1 etc). The search is performed in the ordering of the extension number, regardless of other properties of the certificates. Use the `c_rehash` utility to create the necessary links.

The certificates in `CApath` are only looked up when required, such as when building the certificate chain or when actually performing the verification of a peer certificate.

When looking up CA certificates, the OpenSSL library will first search the certificates in `CAfile`, then those in `CApath`. Certificate matching is done based on the subject name, the key identifier (if present), and the serial number as taken from the certificate to be verified. If these data do not match, the next certificate will be tried. If a first certificate matching the parameters is found, the verification process will be performed; no other certificates for the same parameters will be searched in case of failure.

In server mode, when requesting a client certificate, the server must send the list of CAs from which it will accept client certificates. This listis not influenced by the contents of CAfile or CApath and must explicitely be set using the `SSL_CTX_set_client_CA_list()` family of functions.

When building its own certificate chain, an OpenSSL client/server will try to fill in missing certificates from CAfile/CApath, if the certificate chain was not explicitly specified. (See `SSL_CTX_add_extra_chain_cert()` and `SSL_CTX_use_certificate()`.)

RESTRICTIONS

If several CA certificates matching the name, key identifier, and serial number condition are available, only the first one will be examined. This may lead to unexpected results if the same CA certificate is available with different expiration dates. If a "certificate expired" verification error occurs, no other certificate will be searched. Do no mix expired certificates with valid certificates.

RETURN VALUES

The following return values can occur:

0

The operation failed because `CAfile` and `CApath` are NULL or the processing at one of the locations specified failed. Check the error stack to find out the reason.

1

The operation succeeded.

EXAMPLES

Generate a CA certificate file with descriptive text from the CA certificates `ca1.pem ca2.pem ca3.pem`:

```
#!/bin/sh
rm CAfile.pem
for i in ca1.pem ca2.pem ca3.pem ; do
  openssl x509 -in $i -text >> CAfile.pem
done
```

Prepare the directory `/some/where/certs` containing several CA certificates for use as `CApath`:

```
cd /some/where/certs
c_rehash
```

SEE ALSO

Functions: *ssl*, *SSL_CTX_set_client_CA_list*, *SSL_get_client_CA_list*, *SSL_CTX_use_certificate*, *SSL_CTX_add_extra_chain_cert*

SSL_CTX_new

NAME

SSL_CTX_new – Create a new SSL_CTX object as framework for TLS/SSL enabled functions

SYNOPSIS

```
#include <openssl/ssl.h>
SSL_CTX *SSL_CTX_new(
        SSL_METHOD *method
);
```

DESCRIPTION

The `SSL_CTX_new()` function creates a new `SSL_CTX` object as framework to establish TLS/SSL enabled connections.

NOTES

The `SSL_CTX` object uses `method` as connection method. The methods exist in a generic type (for client and server use), a server only type, and a client only type. The `method` can be of the following types:

`SSLv2_method(void), SSLv2_server_method(void), SSLv2_client_method(void)`

A TLS/SSL connection established with these methods will only understand the SSLv2 protocol. A client will send out SSLv2 client hello messages and will also indicate that it only understand SSLv2. A server will only understand SSLv2 client hello messages.

`SSLv3_method(void), SSLv3_server_method(void), SSLv3_client_method(void)`

A TLS/SSL connection established with these methods will only understand the SSLv3 protocol. A client will send out SSLv3 client hello messages and will indicate that it only understands SSLv3. A server will only understand SSLv3 client hello messages. This especially means, that it will not understand SSLv2 client hello messages which are widely used for compatibility reasons. See `SSLv23_*_method()`.

`TLSv1_method(void), TLSv1_server_method(void), TLSv1_client_method(void)`

A TLS/SSL connection established with these methods will only understand the TLSv1 protocol. A client will send out TLSv1 client hello messages and will indicate that it only understands TLSv1. A server will only understand TLSv1 client hello messages. This especially means, that it will not understand SSLv2 client hello messages which are widely used for compatibility reasons, see SSLv23_*_method(). It will also not understand SSLv3 client hello messages.

`SSLv23_method(void), SSLv23_server_method(void), SSLv23_client_method(void)`

A TLS/SSL connection established with these methods will understand the SSLv2, SSLv3, and TLSv1 protocol. A client will send out SSLv2 client hello messages and will indicate that it also understands SSLv3 and TLSv1. A server will understand SSLv2, SSLv3, and TLSv1 client hello messages. This is the best choice when compatibility is a concern.

If a generic method is used, it is necessary to explicitly set client or server mode with the `SSL_set_connect_state()` or `SSL_set_accept_state()` functions.

The list of protocols available can later be limited using the `SSL_OP_NO_SSLv2, SSL_OP_NO_SSLv3, SSL_OP_NO_TLSv1` options of the `SSL_CTX_set_options()` or `SSL_set_options()` functions. Using these options, it is possible to choose the `SSLv23_server_method()` function, for example, and be able to negotiate with all possible clients, but to only allow newer protocols like SSLv3 or TLSv1.

The `SSL_CTX_new()` function initializes the list of ciphers, the session cache setting, the callbacks, the keys and certificates, and the options to its default values.

RETURN VALUES

The following return values can occur:

NULL

> The creation of a new SSL_CTX object failed. Check the error stack to determine the reason.

Pointer to an SSL_CTX object

> The return value points to an allocated SSL_CTX object.

SEE ALSO

Functions: *SSL_CTX_free*, *SSL_accept*, *ssl*, *SSL_set_connect_state*

SSL_CTX_sess_number

NAME

SSL_CTX_sess_number, SSL_CTX_sess_connect, SSL_CTX_sess_connect_good, SSL_CTX_sess_connect_renegotiate, SSL_CTX_sess_accept, SSL_CTX_sess_accept_good, SSL_CTX_sess_accept_renegotiate, SSL_CTX_sess_cb_hits, SSL_CTX_sess_hits, SSL_CTX_sess_misses, SSL_CTX_sess_timeouts, SSL_CTX_sess_cache_full – Obtain session cache statistics

SYNOPSIS

```
#include <openssl/ssl.h>

long SSL_CTX_sess_number(
        SSL_CTX *ctx
);

long SSL_CTX_sess_connect(
        SSL_CTX *ctx
);

long SSL_CTX_sess_connect_good(
        SSL_CTX *ctx
);

long SSL_CTX_sess_connect_renegotiate(
       SSL_CTX *ctx
);

long SSL_CTX_sess_accept(
        SSL_CTX *ctx
);

long SSL_CTX_sess_accept_good(
        SSL_CTX *ctx
);

long SSL_CTX_sess_accept_renegotiate(
       SSL_CTX *ctx
);

long SSL_CTX_sess_hits(
       SSL_CTX *ctx
);

long SSL_CTX_sess_cb_hits(
        SSL_CTX *ctx
);

long SSL_CTX_sess_misses(
        SSL_CTX *ctx
);
```

```
long SSL_CTX_sess_timeouts(
        SSL_CTX *ctx
);
long SSL_CTX_sess_cache_full(
        SSL_CTX *ctx
);
```

DESCRIPTION

The `SSL_CTX_sess_number()` function returns the current number of sessions in the internal session cache.

The `SSL_CTX_sess_connect()` function returns the number of started SSL/TLS handshakes in client mode.

The `SSL_CTX_sess_connect_good()` function returns the number of successfully established SSL/TLS sessions in client mode.

The `SSL_CTX_sess_connect_renegotiate()` function returns the number of start renegotiations in client mode.

The `SSL_CTX_sess_accept()` function returns the number of started SSL/TLS handshakes in server mode.

The `SSL_CTX_sess_accept_good()` function returns the number of successfully established SSL/TLS sessions in server mode.

The `SSL_CTX_sess_accept_renegotiate()` function returns the number of start renegotiations in server mode.

The `SSL_CTX_sess_hits()` function returns the number of successfully reused sessions. In client mode a session set with the `SSL_set_session()` function successfully reused is counted as a hit. In server mode a session successfully retrieved from internal or external cache is counted as a hit.

The `SSL_CTX_sess_cb_hits()` function returns the number of successfully retrieved sessions from the external session cache in server mode.

The `SSL_CTX_sess_misses()` function returns the number of sessions proposed by clients that were not found in the internal session cache in server mode.

The `SSL_CTX_sess_timeouts()` function returns the number of sessions proposed by clients and either found in the internal or external session cache in server mode, but that were invalid due to timeout. These sessions are not included in the `SSL_CTX_sess_hits()` count.

The `SSL_CTX_sess_cache_full()` function returns the number of sessions that were removed because the maximum session cache size was exceeded.

RETURN VALUES

See the Description section.

SEE ALSO

Functions: *ssl*, *SSL_set_session*, *SSL_CTX_set_session_cache_mode* *SSL_CTX_sess_set_cache_size*

SSL_CTX_sess_set_cache_size

NAME

SSL_CTX_sess_set_cache_size, SSL_CTX_sess_get_cache_size – Manipulate session cache size

SYNOPSIS

```
#include <openssl/ssl.h>
long SSL_CTX_sess_set_cache_size(
        SSL_CTX *ctx, long t
);
long SSL_CTX_sess_get_cache_size(
        SSL_CTX *ctx
);
```

DESCRIPTION

The `SSL_CTX_sess_set_cache_size()` function sets the size of the internal session cache of context `ctx` to `t`.

The `SSL_CTX_sess_get_cache_size()` function returns the currently valid session cache size.

NOTES

The internal session cache size is `SSL_SESSION_CACHE_MAX_SIZE_DEFAULT`, 1024*20, so that up to 20,000 sessions can be held. This size can be modified using the `SSL_CTX_sess_set_cache_size()` function. A special case is the size 0, which is used for unlimited size.

When the maximum number of sessions is reached, no new sessions are added to the cache. New space may be added by calling the `SSL_CTX_flush_sessions()` function to remove expired sessions.

If the size of the session cache is reduced and more sessions are in the session cache, an old session will be removed when a new session is added. This removal is not synchronized with the expiration of sessions.

RETURN VALUES

The `SSL_CTX_sess_set_cache_size()` function returns the previously valid size.

The `SSL_CTX_sess_get_cache_size()` function returns the currently valid size.

SEE ALSO

Functions: *ssl*, *SSL_CTX_set_session_cache_mode*, *SSL_CTX_sess_number*, *SSL_CTX_flush_sessions*

SSL_CTX_sess_set_get_cb

NAME

SSL_CTX_sess_set_get_cb, SSL_CTX_sess_set_new_cb, SSL_CTX_sess_set_remove_cb, SSL_CTX_sess_get_new_cb, SSL_CTX_sess_get_remove_cb, SSL_CTX_sess_get_get_cb – Provide callback functions for server side external session caching

SYNOPSIS

```
#include <openssl/ssl.h>

void SSL_CTX_sess_set_new_cb(
        SSL_CTX *ctx, int (*new_session_cb)(SSL *, SSL_SESSION *)
);

void SSL_CTX_sess_set_remove_cb(
        SSL_CTX *ctx, void (*remove_session_cb)(SSL_CTX *ctx, SSL_SESSION *)
);

void SSL_CTX_sess_set_get_cb(
        SSL_CTX *ctx, SSL_SESSION (*get_session_cb)(SSL *, unsigned char *, int, int *)
);

int
       *SSL_CTX_sess_get_new_cb(SSL_CTX *ctx))(struct ssl_st *ssl, SSL_SESSION *sess);
       void (*SSL_CTX_sess_get_remove_cb(SSL_CTX *ctx))(struct ssl_ctx_st *ctx,
       SSL_SESSION *sess
);

SSL_SESSION *(
       *SSL_CTX_sess_get_get_cb(SSL_CTX *ctx))(struct ssl_st *ssl, unsigned char *data,
       int len, int *copy
);

int
       *new_session_cb)(struct ssl_st *ssl, SSL_SESSION *sess
);

void
       *remove_session_cb)(struct ssl_ctx_st *ctx, SSL_SESSION *sess
);

SSL_SESSION *(
       *get_session_cb)(struct ssl_st *ssl, unsigned char *data, int len, int *copy
);
```

DESCRIPTION

The `SSL_CTX_sess_set_get_cb()` function sets the callback function which is called whenever an SSL/TLS client proposes to resume a session but the session could not be found in the internal session cache (see *SSL_CTX_set_session_cache_mode*). (SSL/TLS server only.)

The `SSL_CTX_sess_set_new_cb()` function sets the callback function, which is automatically called whenever a new session is negotiated.

The `SSL_CTX_sess_set_remove_cb()` function sets the callback function, which is automatically called whenever a session is removed by the SSL engine, because it is considered faulty or the session has become obsolete because of exceeding the timeout value.

The `SSL_CTX_sess_get_new_cb()`, `SSL_CTX_sess_get_remove_cb()`, and `SSL_CTX_sess_get_get_cb()` functions retrieve the function pointers of the provided callback functions. If a callback function has not been set, the NULL pointer is returned.

NOTES

In order to allow external session caching, synchronization with the internal session cache is realized via callback functions. Inside these callback functions, session can be saved to disk or put into a database using the *d2i_SSL_SESSION* interface.

The `new_session_cb()` function is called whenever a new session has been negotiated and session caching is enabled (see *SSL_CTX_set_session_cache_mode*). The `new_session_cb()` function is passed the ssl connection and the ssl session `sess`. If the callback returns `0`, the session will be removed immediately.

The `remove_session_cb()` function is called whenever the SSL engine removes a session from the internal cache. This happens if the session is removed because it is expired or when a connection was not shutdown cleanly. The `remove_session_cb()` function is passed the `ctx` and the ssl session `sess`. It does not provide any feedback.

The `get_session_cb()` function is only called on SSL/TLS servers with the session id proposed by the client. The `get_session_cb()` function is always called when session caching is disabled. The `get_session_cb()` function is passed the `ssl` connection, the session id of length `length` at the memory location `data`. With the parameter `copy` the callback can require the SSL engine to increment the reference count of the SSL_SESSION object.

SEE ALSO

Functions: *ssl*, *d2i_SSL_SESSION*, *SSL_CTX_set_session_cache_mode*, *SSL_CTX_flush_sessions*

SSL_CTX_sessions

NAME

SSL_CTX_sessions – Access internal session cache

SYNOPSIS

```
#include <openssl/ssl.h>
struct lhash_st *SSL_CTX_sessions(
       SSL_CTX *ctx
);
```

DESCRIPTION

The `SSL_CTX_sessions()` function returns a pointer to the lhash databases containing the internal session cache for `ctx`.

NOTES

The sessions in the internal session cache are kept in an *lhash* type database. It is possible to directly access this database. In parallel, the sessions form a linked list which is maintained separately from the *lhash* operations, so that the database must not be modified directlyexcept with the `SSL_CTX_add_session()` family of functions.

SEE ALSO

Functions: *ssl*, *lhash*, *SSL_CTX_add_session*, *SSL_CTX_set_session_cache_mode*

SSL_CTX_set_cert_store

NAME

SSL_CTX_set_cert_store – Set the X509_STORE structure in the SSL_CTX structure

SYNOPSIS

```
#include <openssl/ssl.h>
void SSL_CTX_set_cert_store(
         SSL_CTX *ctx)
        (X509_STORE  *store
);
```

DESCRIPTION

The `SSL_CTX_set_cert_store()` function sets the X509_STORE structure in the SSL_CTX structure. An X509_STORE structure holds information for certificate verification including cache of trusted certificate, external lookup methods and a pointer to a certificate verification function.

SEE ALSO

Functions: *SSL_CTX_get_cert_store*

SSL_CTX_set_cert_verify_cb

NAME

SSL_CTX_set_cert_verify_cb – Set a callback which will be called for certificate verification

SYNOPSIS

```
#include <openssl/ssl.h>
void SSL_CTX_set_cert_verify_cb(
         SSL_CTX *ctx)
        (int (*cb )())
        (char *arg
);
```

DESCRIPTION

The `SSL_CTX_set_cert_verify_cb()` sets a callback function which will be called when a certificate or certificate chain to be verified is passed. The callback function is an alernative to the default built-in function, `X509_verify_cert()`.

NOTES

The `SSL_CTX_set_cert_verify_cb()` function is not available on OpenVMS.

SEE ALSO

SSL_CTX_set_verify

SSL_CTX_set_cipher_list

NAME

SSL_CTX_set_cipher_list, SSL_set_cipher_list – Choose list of available SSL_CIPHERs

SYNOPSIS

```
#include <openssl/ssl.h>
int SSL_CTX_set_cipher_list(
        SSL_CTX *ctx, const char *str
);
int SSL_set_cipher_list(
        SSL *ssl, const char *str
);
```

DESCRIPTION

The `SSL_CTX_set_cipher_list()` function sets the list of available ciphers for `ctx` using the control string `str`. The format of the string is described in *ciphers*. The list of ciphers is inherited by all `ssl` objects created from `ctx`.

The `SSL_set_cipher_list()` function sets the list of ciphers only for `ssl`.

NOTES

The control string `str` should be universally usable and not depend on details of the library configuration (ciphers compiled in). Thus no syntax checking takes place. Items that are not recognized, because the corresponding ciphers are not compiled in or because they are mistyped, are ignored. Failure is only flagged if no ciphers could be collected.

Inclusion of a cipher to be used into the list is a necessary condition. On the client side, the inclusion into the list is also sufficient. On the server side, additional restrictions apply. All ciphers have additional requirements. ADH ciphers do not need a certificate, but DH-parameters must have been set. All other ciphers need a corresponding certificate and key. An RSA cipher can only be chosen when an RSA certificate is available. The respective is valid for DSA ciphers. Ciphers using EDH need a certificate, key and DH parameters.

RETURN VALUES

The `SSL_CTX_set_cipher_list()` and `SSL_set_cipher_list()` functions return 1 if any cipher could be selected and 0 on complete failure.

SEE ALSO

Commands: *ciphers*

Functions: *ssl*, *SSL_get_ciphers*, *SSL_CTX_use_certificate*

SSL_CTX_set_client_CA_list

NAME

SSL_CTX_set_client_CA_list, SSL_set_client_CA_list, SSL_CTX_add_client_CA, SSL_add_client_CA – Set list of CAs sent to the client when requesting a client certificate

SYNOPSIS

```
#include <openssl/ssl.h>

void SSL_CTX_set_client_CA_list(
        SSL_CTX *ctx, STACK_OF(X509_NAME) *list
);

void SSL_set_client_CA_list(
        SSL *s, STACK_OF(X509_NAME) *list
);

int SSL_CTX_add_client_CA(
        SSL_CTX *ctx, X509 *cacert
);

int SSL_add_client_CA(
       SSL *ssl, X509 *cacert
);
```

DESCRIPTION

The `SSL_CTX_set_client_CA_list()` function sets the `list` of CAs sent to the client when requesting a client certificate for `ctx`.

The `SSL_set_client_CA_list()` function sets the `list` of CAs sent to the client when requesting a client certificate for the chosen `ssl`, overriding the setting valid for `ssl`'s SSL_CTX object.

The `SSL_CTX_add_client_CA()` function adds the CA name extracted from `cacert` to the list of CAs sent to the client when requesting a client certificate for `ctx`.

The `SSL_add_client_CA()` function adds the CA name extracted from `cacert` to the list of CAs sent to the client when requesting a client certificate for the chosen `ssl`, overriding the setting valid for `ssl`'s SSL_CTX object.

NOTES

When a TLS/SSL server requests a client certificate (see `SSL_CTX_set_verify_options()`), it sends a list of CAs, for which it will accept certificates, to the client.

This list can be explicitly set using the `SSL_CTX_set_client_CA_list()` function for `ctx` and the `SSL_set_client_CA_list()` function for the specific `ssl`. The list specified overrides the previous setting. The CAs listed do not become trusted (`list` only contains the names, not the complete certificates); use the `SSL_CTX_load_verify_locations()` function to additionally load them for verification.

If the list of acceptable CAs is compiled in a file, the `SSL_load_client_CA_file()` function can be used to help import the necessary data.

The SSL_CTX_add_client_CA() and SSL_add_client_CA() functions can be used to add additional items to the list of client CAs. If no list was specified before using SSL_CTX_set_client_CA_list() or SSL_set_client_CA_list(), a new client CA list for ctx or ssl (as appropriate) is opened.

These functions are only useful for TLS/SSL servers.

RETURN VALUES

The SSL_CTX_set_client_CA_list() and SSL_set_client_CA_list() functions do not return diagnostic information.

The SSL_CTX_add_client_CA() and SSL_add_client_CA() functions have the following return values:

- 1

  The operation succeeded.

- 0

  A failure while manipulating the STACK_OF(X509_NAME) object occurred or the X509_NAME could not be extracted from cacert. Check the error stack to find the reason.

EXAMPLES

Scan all certificates in CAfile and list them as acceptable CAs:

```
SSL_CTX_set_client_CA_list (ctx,SSL_load_client_CA_file(CAfile));
```

SEE ALSO

Functions: *ssl*, *SSL_get_client_CA_list*, *SSL_load_client_CA_file*, *SSL_CTX_load_verify_locations*

SSL_CTX_set_def_verify_paths

NAME

SSL_CTX_set_def_verify_paths – Sets default file path and file name of trusted CA certificate

SYNOPSIS

```
#include <openssl/ssl.h>
int SSL_CTX_set_def_verify_paths(
        SSL_CTX *ctx
);
```

DESCRIPTION

The `SSL_CTX_set_def_verify_paths()` function sets pre-compiled default CA path and file name for certificate verification. The default CA file path and name are defined as follows:

#define X509_CERT_DIR	"SSLCERTS:"
#define X509_CERT_FILE	"SSLCERTS: cert.pem"

NOTES

The `SSL_CTX_set_default_verify_paths()` function is not available on OpenVMS.

RETURN VALUES

The following return values can occur:

0

The operation failed.

1

The operation succeeded.

SEE ALSO

Functions: *SSL_CTX_load_verify_locations*, *SSL_CTX_set_verify*

SSL_CTX_set_default_passwd_cb

NAME

SSL_CTX_set_default_passwd_cb, SSL_CTX_set_default_passwd_cb_userdata – Set password callback for encrypted PEM file handling

SYNOPSIS

```
#include <openssl/ssl.h>
void SSL_CTX_set_default_passwd_cb(
        SSL_CTX *ctx, pem_password_cb *cb
);
void SSL_CTX_set_default_passwd_cb_userdata(
        SSL_CTX *ctx, void *u
);
int pem_passwd_cb(
        char *buf, int size, int rwflag, void *userdata
);
```

DESCRIPTION

The `SSL_CTX_set_default_passwd_cb()` function sets the default password callback called when loading or storing a PEM certificate with encryption.

The `SSL_CTX_set_default_passwd_cb_userdata()` function sets a pointer to `userdata` which will be provided to the password callback on invocation.

The `pem_passwd_cb()` function, which must be provided by the application, hands back the password to be used during decryption. On invocation a pointer to `userdata` is provided. The `pem_passwd_cb()` must write the password into the provided buffer `buf` which is of size `size`. The actual length of the password must be returned to the calling function. The `rwflag` indicates whether the callback is used for reading/decryption (rwflag=0) or writing/encryption (rwflag=1).

NOTES

When loading or storing private keys, a password might be supplied to protect the private key. The way this password can be supplied might depend on the application. If only one private key is handled, it can be practical to have `pem_passwd_cb()` handle the password dialog interactively. If several keys have to be handled, it can be practical to ask for the password once, then keep it in memory and use it several times. In the last case, the password could be stored into the `userdata` storage and the `pem_passwd_cb()` only returns the password already stored.

Other items in PEM formatting (certificates) can also be encrypted. It is not usual, as certificate information is considered public.

RETURN VALUES

The `SSL_CTX_set_default_passwd_cb()` and `SSL_CTX_set_default_passwd_cb_userdata()` functions do not provide diagnostic information.

EXAMPLES

The following example returns the password provided as `userdata` to the calling function. The password is considered to be a '\0' terminated string. If the password does not fit into the buffer, the password is truncated.

```
int pem_passwd_cb(char *buf, int size, int rwflag, void *password)
{
 strncpy(buf, (char *)(password), size);
 buf[size - 1] = '\0';
 return(strlen(buf));
}
```

SEE ALSO

Functions: *ssl*, *SSL_CTX_use_certificate*

SSL_CTX_set_info_callback

NAME

SSL_CTX_set_info_callback, SSL_CTX_get_info_callback, SSL_set_info_callback, SSL_get_info_callback – Handle information callback for SSL connections

SYNOPSIS

```
#include <openssl/ssl.h>
void SSL_CTX_set_info_callback(
         SSL_CTX *ctx)
        (void ( *callback)()
);
void (*SSL_CTX_get_info_callback)(
        SSL_CTX *ctx))(
);
void SSL_set_info_callback(
         SSL *ssl)
        (void (*callback )()
);
void (*SSL_get_info_callback)(
         SSL *ssl))(
);
int
        *callback)(
);
```

DESCRIPTION

The `SSL_CTX_set_info_callback()` function sets the callback function that can be used to obtain state information for SSL objects created from ctx during connection setup and use. The setting for ctx is overridden from the setting for a specific SSL object, if specified. When callback is NULL, no callback function is used.

The `SSL_set_info_callback()` function sets the callback function, that can be used to obtain state information for ssl during connection setup and use. When callback is NULL, the callback setting currently valid for ctx is used.

The `SSL_CTX_get_info_callback()` function returns a pointer to the currently set information callback function for ctx.

The `SSL_get_info_callback()` function returns a pointer to the currently set information callback function for ssl.

NOTES

When setting up a connection and during use, it is possible to obtain state information from the SSL/TLS engine. When set, an information callback function is called whenever the state changes, an alert appears, or an error occurs.

The callback function is called as `callback(SSL *ssl, int where, int ret)`. The `where` argument specifies information about where (in which context) the callback function was called. If `ret` is 0, an error condition occurred. If an alert is handled, SSL_CB_ALERT is set and `ret` specifies the alert information.

The `where` argument is a bitmask made up of the following bits:

- SSL_CB_LOOP

  Callback has been called to indicate state change inside a loop.

- SSL_CB_EXIT

  Callback has been called to indicate error exit of a handshake function. (May be soft error with retry option for nonblocking setups.)

- SSL_CB_READ

  Callback has been called during read operation.

- SSL_CB_WRITE

  Callback has been called during write operation.

- SSL_CB_ALERT

  Callback has been called due to an alert being sent or received

- SSL_CB_READ_ALERT (SSL_CB_ALERT|SSL_CB_READ)
- SSL_CB_WRITE_ALERT (SSL_CB_ALERT|SSL_CB_WRITE)
- SSL_CB_ACCEPT_LOOP (SSL_ST_ACCEPT|SSL_CB_LOOP)
- SSL_CB_ACCEPT_EXIT (SSL_ST_ACCEPT|SSL_CB_EXIT)
- SSL_CB_CONNECT_LOOP (SSL_ST_CONNECT|SSL_CB_LOOP)
- SSL_CB_CONNECT_EXIT (SSL_ST_CONNECT|SSL_CB_EXIT)
- SSL_CB_HANDSHAKE_START

  Callback has been called because a new handshake is started.

- SSL_CB_HANDSHAKE_DONE

  Callback has been called because a handshake is finished.

The current state information can be obtained using the *SSL_state_string* family of functions. The `ret` information can be evaluated using the *SSL_alert_type_string* family of functions.

RETURN VALUES

The `SSL_set_info_callback()` function does not provide diagnostic information. The `SSL_get_info_callback()` function returns the current setting.

EXAMPLES

The following example callback function prints state strings, information about alerts being handled and error messages to the `bio_err` BIO:

```
void apps_ssl_info_callback(SSL *s, int where, int ret)
        {
        const char *str;
        int w;
        w=3Dwhere& ~SSL_ST_MASK;
        if (w & SSL_ST_CONNECT) str=3D"SSL_connect";
        else if (w & SSL_ST_ACCEPT) str=3D"SSL_accept";
        else str=3D"undefined";
        if (where & SSL_CB_LOOP)
                {

BIO_printf(bio_err,"%s:%s\n",str,SSL_state_string_long(s));
                }

        else if (where & SSL_CB_ALERT)
                {
                str=3D(where & SSL_CB_READ)?"read":"write";
                BIO_printf(bio_err,"SSL3 alert %s:%s:%s\n",
                        str,
                        SSL_alert_type_string_long(ret),
                        SSL_alert_desc_string_long(ret));
                }

        else if (where & SSL_CB_EXIT)
                {
                if (ret =3D=3D 0)
                        BIO_printf(bio_err,"%s:failed in %s\n",
                                str,SSL_state_string_long(s));
                else if (ret < 0)
                        {
                        BIO_printf(bio_err,"%s:error in %s\n",
                                str,SSL_state_string_long(s));
                        }
                }
        }
```

SEE ALSO

Functions: *ssl*, *SSL_state_string*, *SSL_alert_type_string*

SSL_CTX_set_mode

NAME

SSL_CTX_set_mode, SSL_set_mode, SSL_CTX_get_mode, SSL_get_mode – Manipulate SSL engine mode

SYNOPSIS

```
#include <openssl/ssl.h>

long SSL_CTX_set_mode(
        SSL_CTX *ctx, long mode
);

long SSL_set_mode(
        SSL *ssl, long mode
);

long SSL_CTX_get_mode(
        SSL_CTX *ctx
);

long SSL_get_mode(
        SSL *ssl
);
```

DESCRIPTION

The `SSL_CTX_set_mode()` function adds the mode set via bitmask in `mode` to `ctx`. Options already set before are not cleared.

The `SSL_set_mode()` function adds the mode set via bitmask in `mode` to `ssl`. Options already set before are not cleared.

The `SSL_CTX_get_mode()` function returns the mode set for `ctx`.

The `SSL_get_mode()` function returns the mode set for `ssl`.

NOTES

The following mode changes are available:

SSL_MODE_ENABLE_ PARTIAL_WRITE

Allow `SSL_write(..., n)` to return r with 0 < r < n (i.e. report success when just a single record has been written). When not set (the default), `SSL_write()` will only report success once the complete chunk was written.

SSL_MODE_ACCEPT_ MOVING_WRITE_ BUFFER

Make it possible to retry `SSL_write()` with changed buffer location (the buffer contents must stay the same). This is not the default to avoid the misconception that non-blocking `SSL_write()` behaves like non-blocking write().

SSL_MODE_AUTO_ RETRY

Never bother the application with retries if the transport is blocking. If a renegotiation takes place during normal operation, a `SSL_read()` or `SSL_write()` would return with -1 and indicate the need to retry with SSL_ERROR_WANT_READ. In a non-blocking environment applications must be prepared to handle incomplete read/write operations. In a blocking environment, applications are not always prepared to deal with read/write operations returning without success report. The SSL_MODE_AUTO_RETRY flag will cause read/write operations to return only after the handshake and successful completion.

RETURN VALUES

The `SSL_CTX_set_mode()` and `SSL_set_mode()` functions return the new mode bitmask after adding `mode`.

The `SSL_CTX_get_mode()` and `SSL_get_mode()` functions return the current bitmask.

HISTORY

SSL_MODE_AUTO_RETRY was added in OpenSSL 0.9.6.

SEE ALSO

Functions: *ssl*, *SSL_read*, *SSL_write*

SSL_CTX_set_options

NAME

SSL_CTX_set_options, SSL_set_options, SSL_CTX_get_options, SSL_get_options – Manipulate SSL engine options

SYNOPSIS

```
#include <openssl/ssl.h>
long SSL_CTX_set_options(
        SSL_CTX *ctx, long options
);
long SSL_set_options(
       SSL *ssl, long options
);
long SSL_CTX_get_options(
        SSL_CTX *ctx
);
long SSL_get_options(
       SSL *ssl
);
```

DESCRIPTION

The `SSL_CTX_set_options()` function adds the options set via bitmask in `options` to `ctx`. Options already set before are not cleared.

The `SSL_set_options()` function adds the options set via bitmask in `options` to `ssl`. Options already set before are not cleared.

The `SSL_CTX_get_options()` function returns the options set for `ctx`.

The `SSL_get_options()` function returns the options set for `ssl`.

NOTES

The behavior of the SSL library can be changed by setting several options. The options are coded as bitmasks and can be combined by a logical `or` operation (`|`). Options can only be added; they can never be reset.

During a handshake, the option settings of the SSL object are used. When a new SSL object is created from a context using `SSL_new()`, the current option setting is copied. Changes to `ctx` do not affect already created SSL objects. The `SSL_clear()` function does not affect the settings.

The following bug workaround options are available:

SSL_OP_MICROSOFT_SESS_ID_BUG

www.microsoft.com , when talking SSLv2, if session-id reuse is performed, the session-id passed back in the server-finished message is different from the one decided upon.

SSL_OP_NETSCAPE_CHALLENGE_BUG

Netscape-Commerce/1.12, when talking SSLv2, accepts a 32-byte challenge but then appears to only use 16 bytes when generating the encryption keys. Using 16 bytes is acceptable, but it should be acceptable also to use 32. According to SSLv3 specifications, you should use 32 bytes for the challenge when operating in SSLv2/v3 compatibility mode, but this breaks the server. So, 16 bytes is preferable.

SSL_OP_NETSCAPE_REUSE_CIPHER_CHANGE_BUG

ssl3.netscape.com:443, first a connection is established with RC4-MD5. If it resumes, you use DES-CBC3-SHA. It should be RC4-MD5 according to 7.6.1.3, 'cipher_suite'.

Netscape-Enterprise/2.01 (https://merchant.netscape.com) has this bug. It only shows up when connecting via SSLv2/v3 then reconnecting via SSLv3. The cipher list changes.

Try connecting with a cipher list of DES-CBC-SHA:RC4-MD5. Each new connection uses RC4-MD5, but a reconnect tries to use DES-CBC-SHA. So, Netscape always takes the first cipher in the cipher list when doing a reconnect.

SSL_OP_SSLREF2_REUSE_CERT_TYPE_BUG

SSL_OP_MICROSOFT_BIG_SSLV3_BUFFER

SSL_OP_MSIE_SSLV2_RSA_PADDING

SSL_OP_SSLEAY_080_CLIENT_DH_BUG

SSL_OP_TLS_D5_BUG

SSL_OP_TLS_BLOCK_PADDING_BUG

SSL_OP_TLS_ROLLBACK_BUG

Disable version rollback attack detection.

During the client key exchange, the client must send the same information about acceptable SSL/TLS protocol levels as during the first hello. Some clients violate this rule by adapting to the server's answer. (Example: the client sends an SSLv2 hello and accepts up to SSLv3.1=TLSv1. The server only understands up to SSLv3. In this case the client must still use the same SSLv3.1=TLSv1 announcement. Some clients step down to SSLv3 with respect to the server's answer and violate the version rollback protection.)

SSL_OP_ALL

All of the above bug workarounds.

We recommended that you use SSL_OP_ALL to enable the bug workaround options.

The following modifying options are available:

SSL_OP_SINGLE_DH_USE

Always create a new key when using temporary DH parameters.

SSL_OP_EPHEMERAL_RSA

Also use the temporary RSA key when doing RSA operations.

SSL_OP_PKCS1_CHECK_1

SSL_OP_PKCS1_CHECK_2

SSL_OP_NETSCAPE_CA_DN_BUG

If you accept a Netscape connection, demand a client cert, have a non-self-signed CA which does not have its CA in netscape, and the browser has a cert, it will crash/hang. Works for 3.x and 4.xbeta

SSL_OP_NON_EXPORT_FIRST

On servers try to use non-export (stronger) ciphers first. This option does not work under all circumstances (in the code it is declared broken).

SSL_OP_NETSCAPE_DEMO_CIPHER_CHANGE_BUG

SSL_OP_NO_SSLv2

Do not use the SSLv2 protocol.

SSL_OP_NO_SSLv3

Do not use the SSLv3 protocol.

SSL_OP_NO_TLSv1

Do not use the TLSv1 protocol.

RETURN VALUES

The `SSL_CTX_set_options()` and `SSL_set_options()` functions return the new options bitmask after adding `options`.

The `SSL_CTX_get_options()` and `SSL_get_options()` functions return the current bitmask.

HISTORY

SSL_OP_TLS_ROLLBACK_BUG was added in OpenSSL 0.9.6.

SEE ALSO

Functions: *ssl*, *SSL_new*, *SSL_clear*

SSL_CTX_set_purpose

NAME

SSL_CTX_set_purpose – Set a purpose value to the SSL_CTX structure

SYNOPSIS

```
#include <openssl/ssl.h>
#include <openssl/x509v3.h> (to use the macros for purpose values)

int SSL_CTX_set_purpose(

       SSL_CTX *s)

       (int purpose

);
```

DESCRIPTION

The `SSL_CTX_set_purpose()` function sets a purpose value in the SSL_CTX structure. The purpose values and their macros are defined in x509v3.h as follows:

```
#define X509_PURPOSE_SSL_CLIENT 1
#define X509_PURPOSE_SSL_SERVER 2
#define X509_PURPOSE_NS_SSL_SERVER 3
#define X509_PURPOSE_SMIME_SIGN 4
#define X509_PURPOSE_SMIME_ENCRYPT 5
#define X509_PURPOSE_CRL_SIGN 6
#define X509_PURPOSE_ANY 7
```

The purpose value must be between 1 and 7. If an out-of-range value is passed, the `SSL_CTX_set_purpose()` function returns 0. Upon success, 1 is returned.

RETURN VALUES

The following return values can occur:

0

Setting the purpose value in the SSL_CTX structure failed.

1

The purpose value was successfully set in the SSL_CTX structure.

SEE ALSO

Functions: *SSL_set_purpose*

SSL_CTX_set_quiet_shutdown

NAME

SSL_CTX_set_quiet_shutdown – Set a value to the quiet-shutdown flag in the SSL_CTX data structure

SYNOPSIS

```
#include <openssl/ssl.h>
void SSL_CTX_set_quiet_shutdown(
        SSL *ssl)
       (int mode
);
```

DESCRIPTION

The `SSL_CTX_set_quiet_shutdown()` function sets a mode of quiet shutdown to the SSL_CTX structure. To turn on the quiet shutdown, mode == 1 need to be passed. mode == 0 turns off the quiet shutdown flag of the SSL_CTX structure. When `SSL_CTX_new()` creates an SSL_CTX structure, 0 is set to the quiet-shutdown flag.

SEE ALSO

Functions: *SSL_CTX_get_quiet_shutdown*, *SSL_get_quiet_shutdown*, *SSL_set_quiet_shutdown*

SSL_CTX_set_session_cache_mode

NAME

SSL_CTX_set_session_cache_mode, SSL_CTX_get_session_cache_mode – Enable or disable session caching

SYNOPSIS

```
#include <openssl/ssl.h>
long SSL_CTX_set_session_cache_mode(
        SSL_CTX ctx, long mode
);
long SSL_CTX_get_session_cache_mode(
        SSL_CTX ctx
);
```

DESCRIPTION

The `SSL_CTX_set_session_cache_mode()` function enables or disables session caching by setting the operational mode for `ctx` to `<mode>`.

The `SSL_CTX_get_session_cache_mode()` function returns the currently used cache mode.

NOTES

The OpenSSL library can store/retrieve SSL/TLS sessions for later reuse. The sessions can be held in memory for each `ctx`. If more than one SSL_CTX object is being maintained, the sessions are unique for each SSL_CTX object.

In order to reuse a session, a client must send the session's id to the server. It can only send one id. The server then decides whether to reuse the session or start the handshake for a new session.

A server will check the session in its internal session storage. If the session is not found in internal storage or internal storage, it is deactivated. The server will try the external storage if available.

Since a client may try to reuse a session intended for use in a different context, the session id context must be set by the server (see *SSL_CTX_set_session_id_context*).

The following session cache modes and modifiers are available:

SSL_SESS_CACHE_OFF

No session caching for client or server takes place.

SSL_SESS_CACHE_CLIENT

Client sessions are added to the session cache. As there is no reliable way for the OpenSSL library to know whether a session should be reused or which session to choose (due to the abstract BIO layer the SSL engine does not have details about the connection), the application must select the session to be reused by using the `SSL_set_session()` function. This option is not activated by default.

SSL_SESS_CACHE_SERVER

Server sessions are added to the session cache. When a client proposes a session be reused, the session is looked up in the internal session cache. If the session is found, the server will try to reuse the session. This is the default.

SSL_SESS_CACHE_BOTH

Enable both SSL_SESS_CACHE_CLIENT and SSL_SESS_CACHE_SERVER at the same time.

SSL_SESS_CACHE_NO_AUTO_CLEAR

Normally the session cache is checked for expired sessions every 255 connections using the `SSL_CTX_flush_sessions()` function. Since this might lead to a delay which cannot be controlled, the automatic flushing can be disabled and the `SSL_CTX_flush_sessions()` can be called explicitly by the application.

SSL_SESS_CACHE_NO_ INTERNAL_LOOKUP

By setting this option, sessions are cached in the internal storage but they are not looked up automatically. If an external session cache is enabled, sessions are looked up in the external cache. As automatic lookup only applies for SSL/TLS servers. The option has no effect on clients.

The default mode is SSL_SESS_CACHE_SERVER.

RETURN VALUES

The `SSL_CTX_set_session_cache_mode()` function returns the previously set cache mode.

The `SSL_CTX_get_session_cache_mode()` function returns the currently set cache mode.

SEE ALSO

Functions: *ssl*, *SSL_set_session*, *SSL_CTX_sess_number*, *SSL_CTX_sess_set_cache_size*, *SSL_CTX_sess_set_get_cb*, *SSL_CTX_set_session_id_context*, *SSL_CTX_set_timeout*, *SSL_CTX_flush_sessions*

SSL_CTX_set_session_id_context

NAME

SSL_CTX_set_session_id_context, SSL_set_session_id_context – Set context within which session can be reused (server side only)

SYNOPSIS

```
#include <openssl/ssl.h>

int SSL_CTX_set_session_id_context(

       SSL_CTX *ctx, const unsigned char *sid_ctx, unsigned int sid_ctx_len

);

int SSL_set_session_id_context(

        SSL *ssl, const unsigned char *sid_ctx, unsigned int sid_ctx_len

);
```

DESCRIPTION

The `SSL_CTX_set_session_id_context()` function sets the context `sid_ctx` of length `sid_ctx_len` within which a session can be reused for the `ctx` object.

The `SSL_set_session_id_context()` function sets the context `sid_ctx` of length `sid_ctx_len` within which a session can be reused for the `ssl` object.

NOTES

Sessions are generated within a certain context. When exporting or importing sessions with `i2d_SSL_SESSION` or `d2i_SSL_SESSION` it is possible, to reimport a session generated from another context (e.g. another application), which might lead to malfunctions. Therefore, each application must set its own session id context `sid_ctx` which is used to distinguish the contexts and is stored in exported sessions. The `sid_ctx` can be any kind of binary data with a given length. For example, it is possible to use the name of the application, the hostname and/or the service name.

The session id context becomes part of the session. The session id context is set by the SSL/TLS server. The `SSL_CTX_set_session_id_context()` and `SSL_set_session_id_context()` functions are therefore only useful on the server side.

OpenSSL clients will check the session id context returned by the server when reusing a session.

The maximum length of the `sid_ctx` is limited to `SSL_MAX_SSL_SESSION_ID_LENGTH`.

RESTRICTIONS

If the session id context is not set on an SSL/TLS server, stored sessions will not be reused. A fatal error will be flagged and the handshake will fail.

If a server returns a different session id context to an OpenSSL client when reusing a session, an error will be flagged and the handshake will fail. OpenSSL servers will always return the correct session id context, because an OpenSSL server checks the session id context before reusing a session.

RETURN VALUES

The `SSL_CTX_set_session_id_context()` and `SSL_set_session_id_context()` functions return the following values:

- 0

  The length `sid_ctx_len` of the session id context `sid_ctx` exceeded the maximum allowed length of `SSL_MAX_SSL_SESSION_ID_LENGTH`. The error is logged to the error stack.

- 1

  The operation succeeded.

SEE ALSO

Functions: *ssl*

SSL_CTX_set_ssl_version

NAME

SSL_CTX_set_ssl_version, SSL_set_ssl_method, SSL_get_ssl_method – Choose a new TLS/SSL method

SYNOPSIS

```
#include <openssl/ssl.h>
int SSL_CTX_set_ssl_version(
        SSL_CTX *ctx, SSL_METHOD *method
);
int SSL_set_ssl_method(
       SSL *s, SSL_METHOD *method); SSL_METHOD *SSL_get_ssl_method(SSL *ssl)
);
```

DESCRIPTION

The `SSL_CTX_set_ssl_version()` function sets a new default TLS/SSL `method` for SSL objects newly created from this `ctx`. SSL objects already created with the `SSL_new()` function are not affected, except when the `SSL_clear()` function is called.

The `SSL_set_ssl_method()` function sets a new TLS/SSL `method` for a particular ssl object. It may be reset, when `SSL_clear()` is called.

The `SSL_get_ssl_method()` function returns a function pointer to the TLS/SSL method set in ssl.

NOTES

The available method choices are described in *SSL_CTX_new*.

When `SSL_clear()` is called and no session is connected to an SSL object, the method of the SSL object is reset to the method currently set in the corresponding SSL_CTX object.

RETURN VALUES

The following return values can occur for the `SSL_CTX_set_ssl_version()` and `SSL_set_ssl_method()` functions:

0

The new choice failed, check the error stack to find out the reason.

1

The operation succeeded.

SEE ALSO

Functions: *SSL_CTX_new*, *SSL_new*, *SSL_clear*, *ssl*, *SSL_set_connect_state*

SSL_CTX_set_timeout

NAME

SSL_CTX_set_timeout, SSL_CTX_get_timeout – Manipulate timeout values for session caching

SYNOPSIS

```
#include <openssl/ssl.h>
long SSL_CTX_set_timeout(
        SSL_CTX *ctx, long t
);
long SSL_CTX_get_timeout(
        SSL_CTX *ctx
);
```

DESCRIPTION

The `SSL_CTX_set_timeout()` function sets the timeout for newly created sessions for `ctx` to `t`. The timeout value `t` must be given in seconds.

The `SSL_CTX_get_timeout()` function returns the currently set timeout value for `ctx`.

NOTES

Whenever a new session is created, it is assigned a maximum lifetime. This lifetime is specified by storing the creation time of the session and the timeout value valid at this time. If the actual time is later than creation time plus timeout, the session is not reused.

Due to this realization, all sessions behave according to the timeout value valid at the time of the session negotiation. Changes of the timeout value do not affect already established sessions.

The expiration time of a single session can be modified using the `SSL_SESSION_get_time()` family of functions.

Expired sessions are removed from the internal session cache whenever `SSL_CTX_flush_sessions()` is called, either directly by the application or automatically (see *SSL_CTX_set_session_cache_mode*).

The default value for session timeout is 300 seconds.

RETURN VALUES

The `SSL_CTX_set_timeout()` function returns the previously set timeout value.

`SSL_CTX_get_timeout()` function returns the currently set timeout value.

SEE ALSO

Functions: *ssl*, *SSL_CTX_set_session_cache_mode*, *SSL_SESSION_get_time*, *SSL_CTX_flush_sessions*

SSL_CTX_set_tmp_dh_callback

NAME

SSL_CTX_set_tmp_dh_callback, SSL_CTX_set_tmp_dh, SSL_set_tmp_dh_callback, SSL_set_tmp_dh – Handle DH keys for ephemeral key exchange

SYNOPSIS

```
#include <openssl/ssl.h>
void SSL_CTX_set_tmp_dh_callback(
         SSL_CTX *ctx)
        (DH *(*tmp_dh_callback)(SSL  *ssl)
        (int is_export)
        (int keylength)
);
long SSL_CTX_set_tmp_dh(
         SSL_CTX *ctx)
        (DH *dh
);
void SSL_set_tmp_dh_callback(
         SSL_CTX *ctx)
        (DH *(*tmp_dh_callback)(SSL  *ssl)
        (int is_export)
        (int keylength)
);
long SSL_set_tmp_dh(
        SSL  *ssl)
        (DH *dh))
        (DH *(*tmp_dh_callback)(SSL *ssl )
        (int is_export)
        (int keylength)
);
```

DESCRIPTION

The `SSL_CTX_set_tmp_dh_callback()` function sets the callback function for ctx to be used when DH parameters are required to tmp_dh_callback. The callback is inherited by all ssl objects created from ctx.

The `SSL_CTX_set_tmp_dh()` function sets *DH* parameters to be used to be dh. The key is inherited by all ssl objects created from *ctx*. The `SSL_set_tmp_dh_callback()` function sets the callback only for ssl.

The `SSL_set_tmp_dh()` function sets the parameters only for ssl.

These functions apply to SSL/TLS servers only.

NOTES

When using a cipher with RSA authentication, an ephemeral DH key exchange can take place. Ciphers with DSA keys always use ephemeral DH keys as well. In these cases, the session data are negotiated using the ephemeral/temporary DH key and the key supplied and certified by the certificate chain is only used for signing. Anonymous ciphers (without a permanent server key) also use ephemeral DH keys.

Using ephemeral DH key exchange yields forward secrecy, as the connection can only be decrypted, when the DH key is known. By generating a temporary DH key inside the server application that is lost when the application is left, it becomes impossible for an attacker to decrypt past sessions, even if he gets hold of the normal (certified) key, as this key was only used for signing.

In order to perform a DH key exchange the server must use a DH group (DH parameters) and generate a DH key. The server will always generate a new DH key during the negotiation, when the DH parameters are supplied via callback and/or when the `SSL_OP_SINGLE_DH_USE` option of `SSL_CTX_set_options()` is set. It will immediately create a DH key, when DH parameters are supplied via `SSL_CTX_set_tmp_dh()` and `SSL_OP_SINGLE_DH_USE` is not set. In this case, it may happen that a key is generated on initialization without later being needed, while on the other hand the computer time during the negotiation is being saved.

If strong primes were used to generate the DH parameters, it is not necessary to generate a new key for each handshake, but it does improve forward secrecy. If it is not assured, that strong primes were used, `SSL_OP_SINGLE_DH_USE` must be used in order to prevent small subgroup attacks. Always using `SSL_OP_SINGLE_DH_USE` has an impact on the computer time needed during negotiation. Because it is not very large,application authors and users should consider always enabling this option.

Because generating DH parameters is extremely time consuming, an application should not generate the parameters on the fly but supply the parameters. DH parameters can be reused, as the actual key is newly generated during the negotiation. The risk in reusing DH parameters is that an attacker may specialize on a very often used DH group. Applications should therefore generate their own DH parameters during the installation process using the openssl *dHParam* application. In order to reduce the computer time needed for this generation, it is possible to use DSA parameters instead (see *dHParam*), but in this case `SSL_OP_SINGLE_DH_USE` is mandatory.

Application authors can compile in DH parameters. Files `dh512.pem` , `dh1024.pem`, `dh2048.pem`, and `dh4096` in the 'apps' directory of the current version of the OpenSSL distribution contain the 'SKIP' DH parameters, which use safe primes and were generated verifiably pseudo-randomly. These files can be converted into C code using the `C` option of the `dHParam` application. Authors may also generate their own set of parameters using `dHParam`, but a user may not be sure how the parameters were generated. The generation of DH parameters during installation is recommended.

An application may either directly specify the DH parameters or can supply the DH parameters via a callback function. The callback approach has the advantage that the callback may supply DH parameters for different key lengths.

The `tmp_dh_callback` is called with the keylength needed and the `is_export` information. The `is_export` flag is set, when the ephemeral DH key exchange is performed with an export cipher.

RETURN VALUES

The `SSL_CTX_set_tmp_dh_callback()` and `SSL_set_tmp_dh_callback()` functions do not return diagnostic output.

The `SSL_CTX_set_tmp_dh()` and `SSL_set_tmp_dh()` functions do return 1 on success and 0 on failure. Check the error queue to find out the reason of failure.

EXAMPLES

Handle DH parameters for key lengths of 512 and 1024 bits (error handling partly left out):

```
...

DH *dh_512 = NULL;
DH *dh_1024 = NULL;
FILE *paramfile;
...
paramfile = fopen("dh_param_512.pem", "r");
if (paramfile) {
    dh_512 = PEM_read_DHparams(paramfile, NULL, NULL, NULL);
    fclose(paramfile);
}

paramfile = fopen("dh_param_1024.pem", "r");
if (paramfile) {
    dh_1024 = PEM_read_DHparams(paramfile, NULL, NULL, NULL);
    fclose(paramfile);
}
...

DH *get_dh512() { ... }
DH *get_dh1024() { ... }

DH *tmp_dh_callback(SSL *s, int is_export, int keylength)
{
    DH *dh_tmp=NULL;
    switch (keylength) {
    case 512:
      if (!dh_512)
        dh_512 = get_dh512();
    dh_tmp = dh_512;
    break;
    case 1024:
      if (!dh_1024)
        dh_1024 = get_dh1024();
      dh_tmp = dh_1024;
      break;
    default:
      setup_dh_parameters_like_above();
    }
    return(dh_tmp);
}
```

SEE ALSO

Files: *ciphers, dHParam*

Functions: *ssl, SSL_CTX_set_cipher_list, SSL_CTX_set_tmp_rsa_callback, SSL_CTX_set_options*

SSL_CTX_set_tmp_rsa_callback

NAME

SSL_CTX_set_tmp_rsa_callback, SSL_set_tmp_rsa_callback – Set the callback which will be called when a temporary RSA key is required

SYNOPSIS

```
#include <openssl/ssl.h>
void SSL_CTX_set_tmp_rsa_callback(
         SSL_CTX *ctx)
        (RSA *(*cb)(SSL  *ssl)
        (int is_export)
        (int keylength))
void SSL_set_tmp_rsa_callback(
         SSL *ssl)
        (RSA *(*cb)(SSL  *ssl)
        (int is_export)
        (int keylength)
);
```

DESCRIPTION

The `SSL_CTX_set_tmp_dh_callback()` and `SSL_set_tmp_dh_callback()` functions set the callback which will be called when a temporary RSA key is required. The export flag will be set if the reason of needing a temp key is because an export ciphersuite is in use, in which case, keylength will contain the required keylength in bits.

The `SSL_CTX_set_tmp_dh_callback()` function sets the callback to the SSL_CTX structure, and the `SSL_set_tmp_dh_callback()` function sets the callback to the SSL structure.

Creating a temporary RSA key is an expensive CPU operation and is not required by certain cipher suites. The callback is used to delay the creation of the RSA key until it is actually required. Therefore with this API, the problem can start up faster, and the RSA key creation is not performed unless required.

SEE ALSO

Functions: *SSL_set_tmp_dh_callback*, *SSL_CTX_set_tmp_dh_callback*

SSL_CTX_set_trust

NAME

SSL_CTX_set_trust – Set a trust value to the SSL_CTX structure

SYNOPSIS

```
#include <openssl/ssl.h>
#include <openssl/x509.h> (to use the macros of trust values)

int SSL_CTX_set_trust(

        SSL_CTX *s)

        (int trust

);
```

DESCRIPTION

The `SSL_CTX_set_trust()` function sets a trust value in the SSL_CTX structure. The trust values and their macros are defined in x509v3.h as follows:

1. #define X509_TRUST_COMPAT
2. #define X509_TRUST_SSL_CLIENT
3. #define X509_TRUST_SSL_SERVER
4. #define X509_TRUST_EMAIL
5. #define X509_TRUST_OBJECT_SIGN

The trust value must be between 1 and 5. If an out-of-range value is passed, the `SSL_CTX_set_trust()` function returns 0. Upon success, 1 is returned.

RETURN VALUES

The following return values can occur:

1

The trust value was set successfully in the SSL_CTX structure.

0

Setting the trust value in the SSL_CTX structure failed.

SEE ALSO

Functions: *SSL_set_trust*

SSL_CTX_set_verify

NAME

SSL_CTX_set_verify, SSL_set_verify, SSL_CTX_set_verify_depth, SSL_set_verify_depth – Set peer certificate verification parameters

SYNOPSIS

```
#include <openssl/ssl.h>
void SSL_CTX_set_verify(
        SSL_CTX *ctx, int mode, int (*verify_callback)(int, X509- *)
);
void SSL_set_verify(
        SSL *s, int mode, int (*verify_callback)(int, X509_STORE_CTX *)
);
void SSL_CTX_set_verify_depth(
        SSL_CTX *ctx,int depth
);
void SSL_set_verify_depth(
        SSL *s, int depth
);
int verify_callback(
        int preverify_ok, X509_STORE_CTX *x509_ctx
);
```

DESCRIPTION

The `SSL_CTX_set_verify()` function sets the verification flags for `ctx` to be `mode` and specifies the `verify_callback()` function to be used. If no callback function is specified, the NULL pointer can be used for `verify_callback()`.

The `SSL_set_verify()` function sets the verification flags for `ssl` to be `mode` and specifies the `verify_callback()` function to be used. If no callback function is specified, the NULL pointer can be used for `verify_callback()`. In this last case `verify_callback` set specifically for this `ssl` remains. If no special callback was set, the default callback for the underlying `ctx` that was valid at the the time `ssl` was created with the `SSL_new()` function is used.

The `SSL_CTX_set_verify_depth()` function sets the maximum depth for the certificate chain verification that will be allowed for `ctx`.

The `SSL_set_verify_depth()` function sets the maximum depth for the certificate chain verification that will be allowed for `ssl`.

NOTES

The verification of certificates can be controlled by a set of logically or'ed mode flags:

SSL_VERIFY_NONE

Server mode: the server will not send a client certificate request to the client, so the client will not send a certificate.

Client mode: if not using an anonymous cipher (by default disabled), the server will send a certificate which will be checked. The result of the certificate verification process can be checked after the TLS/SSL handshake using the `SSL_get_verify_result()` function. The handshake will be continued regardless of the verification result.

SSL_VERIFY_PEER

Server mode: the server sends a client certificate request to the client. The certificate returned (if any) is checked. If the verification process fails as indicated by verify_callback, the TLS/SSL handshake is immediately terminated with an alert message containing the reason for the verification failure. The behavior can be controlled by the additional SSL_VERIFY_FAIL_IF_NO_PEER_CERT and SSL_VERIFY_CLIENT_ONCE flags.

Client mode: the server certificate is verified. If the verification process fails as indicated by verify_callback, the TLS/SSL handshake is immediately terminated with an alert message containing the reason for the verification failure. If no server certificate is sent, because an anonymous cipher is used, SSL_VERIFY_PEER is ignored.

SSL_VERIFY_FAIL_IF_NO_PEER_CERT

Server mode: if the client did not return a certificate, the TLS/SSL handshake is immediately terminated with a handshake failure alert. This flag must be used together with SSL_VERIFY_PEER.

Client mode: ignored

SSL_VERIFY_CLIENT_ONCE

Server mode: only request a client certificate on the initial TLS/SSL handshake. Do not ask for a client certificate again in case of a renegotiation. This flag must be used together with SSL_VERIFY_PEER.

Client mode: ignored

Either SSL_VERIFY_NONE or SSL_VERIFY_PEER must be set at any time.

The `SSL_CTX_set_verify_depth()` and `SSL_set_verify_depth()` functions set the limit up to which depth certificates in a chain are used during the verification procedure. If the certificate chain is longer than allowed, the certificates above the limit are ignored. Error messages are generated as if these certificates would not be present. Most likely a X509_V_ERR_UNABLE_TO_GET_ISSUER_CERT_LOCALLY will be issued. The depth count is `level 0:peer certificate`, `level 1: CA certificate`, `level 2: higher level CA certificate`, and so on. Setting the maximum depth to 2 allows the levels 0, 1, and 2. The default depth limit is 9, allowing for the peer certificate and additional 9 CA certificates.

The `verify_callback()` function is used to control the behavior when the SSL_VERIFY_PEER flag is set. It must be supplied by the application and receives two arguments: `preverify_ok` indicates, whether the verification of the certificate in question was passed (preverify_ok=1) or not (preverify_ok=0). The `x509_ctx` is a pointer to the complete context used for the certificate chain verification.

The certificate chain is checked starting with the deepest nesting level (the root CA certificate) and worked upward to the peer's certificate. At each level signatures and issuer attributes are checked. Whenever a verification error is found, the error number is stored in `x509_ctx`. The `verify_callback()` function is called with `preverify_ok`=0. By applying `X509_CTX_store_*` functions, `verify_callback` can locate the certificate in question and perform additional steps. If no error is found for a certificate, `verify_callback()` is called with `preverify_ok`=1 before advancing to the next level.

The return value of verify_callback() controls the strategy of the further verification process. If verify_callback() returns 0, the verification process stops with verification failed state. If SSL_VERIFY_PEER is set, a verification failure alert is sent to the peer and the TLS/SSL handshake terminates. If verify_callback() returns 1, the verification process is continued. If verify_callback() always returns 1, the TLS/SSL handshake will never be terminated because of this application experiencing a verification failure. The calling process can, however, retrieve the error code of the last verification error using SSL_get_verify_result() or by maintaining its own error storage managed by verify_callback().

If no verify_callback() is specified, the default callback will be used. Its return value is identical to preverify_ok, so that any verification failure will lead to a termination of the TLS/SSL handshake with an alert message, if SSL_VERIFY_PEER is set.

RESTRICTIONS

In client mode, it is not checked whether the SSL_VERIFY_PEER flag is set, but whether SSL_VERIFY_NONE is not set. This can lead to unexpected behavior, if the SSL_VERIFY_PEER and SSL_VERIFY_NONE are not used as required (one or the other must be set at any time).

The certificate verification depth set with the SSL[_CTX]_verify_depth() function stops the verification at a certain depth. The error message produced will be that of an incomplete certificate chain and not X509_V_ERR_CERT_CHAIN_TOO_LONG as may be expected.

RETURN VALUES

The SSL*_set_verify*() functions do not provide diagnostic information.

EXAMPLES

The following code sequence is an example of the verify_callback() function that will always continue the TLS/SSL handshake regardless of verification failure. The callback realizes a verification depth limit with more informational output.

All verification errors are printed. Information about the certificate chain are printed on request. The example is realized for a server that allows, but not require, client certificates.

The example makes use of the ex_data technique to store application data into or retrieve application data from the SSL structure (see *SSL_get_ex_new_index*, *SSL_get_ex_data_X509_STORE_CTX_idx*).

```
...
typedef struct {
  int verbose_mode;
  int verify_depth;
  int always_continue;
} mydata_t;
int mydata_index;
...
static int verify_callback(int preverify_ok, X509_STORE_CTX *ctx)
{
   char    buf[256];
   X509   *err_cert;
   int     err, depth;
   SSL    *ssl;
   mydata_t *mydata;

   err_cert = X509_STORE_CTX_get_current_cert(ctx);
   err = X509_STORE_CTX_get_error(ctx);
```

```
   depth = X509_STORE_CTX_get_error_depth(ctx);

   /*
    * Retrieve the pointer to the SSL of the connection currently treated
    * and the application specific data stored into the SSL object.
    */

   ssl = X509_STORE_CTX_get_ex_data(ctx, SSL_get_ex_data_X509_STORE_CTX_idx());
   mydata = SSL_get_ex_data(ssl, mydata_index);

   X509_NAME_oneline(X509_get_subject_name(err_cert), buf, 256);

   /*
    * Catch a too long certificate chain. The depth limit set using
    * SSL_CTX_set_verify_depth() is by purpose set to "limit+1" so
    * that whenever the "depth>verify_depth" condition is met, we
    * have violated the limit and want to log this error condition.
    * We must do it here, because the CHAIN_TOO_LONG error would not
    * be found explicitly; only errors introduced by cutting off the
    * additional certificates would be logged.
    */

   if (depth > mydata->verify_depth) {
       preverify_ok = 0;
       err = X509_V_ERR_CERT_CHAIN_TOO_LONG;
       X509_STORE_CTX_set_error(ctx, err);
   }

   if (!preverify_ok) {
       printf("verify error:num=%d:%s:depth=%d:%s\n", err,
                X509_verify_cert_error_string(err), depth, buf);
   }
   else if (mydata->verbose_mode)
   {
       printf("depth=%d:%s\n", depth, buf);
   }

   /*
    * At this point, err contains the last verification error. We can use
    * it for something special
    */
   if (!preverify_ok && (err == X509_V_ERR_UNABLE_TO_GET_ISSUER_CERT)
   {
     X509_NAME_oneline(X509_get_issuer_name(ctx->current_cert), buf, 256);
     printf("issuer= %s\n", buf);
   }

   if (mydata->always_continue)
     return 1;
   else
     return preverify_ok;
}
...

mydata_t mydata;

...
mydata_index = SSL_get_ex_new_index(0, "mydata index", NULL, NULL, NULL);
```

```
...
SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER|SSL_VERIFY_CLIENT_ONCE,
                   verify_callback);
/*
 * Let the verify_callback catch the verify_depth error so that we get
 * an appropriate error in the logfile.
 */
SSL_CTX_set_verify_depth(verify_depth + 1);

/*
 * Set up the SSL specific data into "mydata" and store it into th SSL
 * structure.
 */

mydata.verify_depth = verify_depth; ...
SSL_set_ex_data(ssl, mydata_index, &mydata);

...

SSL_accept(ssl);/* check of success left out for clarity */
if (peer = SSL_get_peer_certificate(ssl))
{
  if (SSL_get_verify_result(ssl) == X509_V_OK)
  {
    /* The client sent a certificate which verified OK */
  }
}
```

SEE ALSO

Functions: *ssl*, *SSL_new*, *SSL_CTX_get_verify_mode*, *SSL_get_verify_result*, *SSL_CTX_load_verify_locations*, *SSL_get_peer_certificate*, *SSL_get_ex_data_X509_STORE_CTX_idx*, *SSL_get_ex_new_index*

SSL_CTX_use_certificate

NAME

SSL_CTX_use_certificate, SSL_CTX_use_certificate_ASN1, SSL_CTX_use_certificate_file, SSL_use_certificate, SSL_use_certificate_ASN1, SSL_use_certificate_file, SSL_CTX_use_certificate_chain_file, SSL_CTX_use_PrivateKey, SSL_CTX_use_PrivateKey_ASN1, SSL_CTX_use_PrivateKey_file, SSL_CTX_use_RSAPrivateKey, SSL_CTX_use_RSAPrivateKey_ASN1, SSL_CTX_use_RSAPrivateKey_file, SSL_use_PrivateKey_file, SSL_use_PrivateKey_ASN1, SSL_use_PrivateKey, SSL_use_RSAPrivateKey, SSL_use_RSAPrivateKey_ASN1, SSL_use_RSAPrivateKey_file, SSL_CTX_check_private_key, SSL_check_private_key – Load certificate and key data

SYNOPSIS

```
#include <openssl/ssl.h>
int SSL_CTX_use_certificate(
        SSL_CTX *ctx, X509 *x
);
int SSL_CTX_use_certificate_ASN1(
        SSL_CTX *ctx, int len, unsigned char *d
);
int SSL_CTX_use_certificate_file(
        SSL_CTX *ctx, const char *file, int type
);
int SSL_use_certificate(
        SSL *ssl, X509 *x
);
int SSL_use_certificate_ASN1(
        SSL *ssl, unsigned char *d, int len
);
int SSL_use_certificate_file(
        SSL *ssl, const char *file, int type
);
int SSL_CTX_use_certificate_chain_file(
       SSL_CTX *ctx, const char *file
);
int SSL_CTX_use_PrivateKey(
        SSL_CTX *ctx, EVP_PKEY *pkey
);
int SSL_CTX_use_PrivateKey_ASN1(
        int pk, SSL_CTX *ctx, unsigned char *d, long len
);
int SSL_CTX_use_PrivateKey_file(
```

```
        SSL_CTX *ctx, const char *file, int type
);
int SSL_CTX_use_RSAPrivateKey(
        SSL_CTX *ctx, RSA *rsa
);
int SSL_CTX_use_RSAPrivateKey_ASN1(
       SSL_CTX *ctx, unsigned char *d, long len
);
int SSL_CTX_use_RSAPrivateKey_file(
       SSL_CTX *ctx, const char *file, int type
);
int SSL_use_PrivateKey(
       SSL *ssl, EVP_PKEY *pkey
);
int SSL_use_PrivateKey_ASN1(
        int pk,SSL *ssl, unsigned char *d, long len
);
int SSL_use_PrivateKey_file(
        SSL *ssl, const char *file, int type
);
int SSL_use_RSAPrivateKey(
        SSL *ssl, RSA *rsa
);
int SSL_use_RSAPrivateKey_ASN1(
        SSL *ssl, unsigned char *d, long len
);
int SSL_use_RSAPrivateKey_file(
        SSL *ssl, const char *file, int type
);
int SSL_CTX_check_private_key(
        SSL_CTX *ctx
);
int SSL_check_private_key(
        SSL *ssl
);
```

DESCRIPTION

These functions load the certificates and private keys into the SSL_CTX or SSL object, respectively.

The SSL_CTX_* class of functions loads the certificates and keys into the SSL_CTX object ctx. The information is passed to SSL objects ssl created from ctx with the SSL_new() function by copying, so that changes applied to ctx do not propagate to already existing SSL objects.

The SSL_* class of functions only loads certificates and keys into a specific SSL object. The specific information is kept when SSL_clear() is called for this SSL object.

The SSL_CTX_use_certificate() function loads the certificate x into ctx, and the SSL_use_certificate() function loads x into ssl. The rest of the certificates needed to form the complete certificate chain can be specified using the SSL_CTX_add_extra_chain_cert() function.

The SSL_CTX_use_certificate_ASN1() function loads the ASN1 encoded certificate from the memory location d (with length len) into ctx. The SSL_use_certificate_ASN1() function loads the ASN1 encoded certificate into ssl.

The SSL_CTX_use_certificate_file() function loads the first certificate stored in file into ctx. The formatting type of the certificate must be specified from the known types SSL_FILETYPE_PEM, SSL_FILETYPE_ASN1. The SSL_use_certificate_file() function loads the certificate from file into ssl.

The SSL_CTX_use_certificate_chain_file() function loads a certificate chain from file into ctx. The certificates must be in PEM format and must be sorted starting with the certificate to the highest level (root CA). There is no corresponding function working on a single SSL object.

The SSL_CTX_use_PrivateKey() function adds pkey as private key to ctx. The SSL_CTX_use_RSAPrivateKey() function adds the private key rsa of type RSA to ctx. The SSL_use_PrivateKey() function adds pkey as private key to ssl. The SSL_use_RSAPrivateKey() function adds rsa as private key of type RSA to ssl.

The SSL_CTX_use_PrivateKey_ASN1() function adds the private key of type pk stored at memory location d (length len) to ctx. The SSL_CTX_use_RSAPrivateKey_ASN1() function adds the private key of type RSA stored at memory location d (length len) to ctx. The SSL_use_PrivateKey_ASN1() and SSL_use_RSAPrivateKey_ASN1() functions add the private key to ssl.

The SSL_CTX_use_PrivateKey_file() function adds the first private key found in file to ctx. The formatting **type** of the certificate must be specified from the known types SSL_FILETYPE_PEM, SSL_FILETYPE_ASN1. The SSL_CTX_use_RSAPrivateKey_file() function adds the first private RSA key found in file to ctx. The SSL_use_PrivateKey_file() function adds the first private key found in file to ssl. The SSL_use_RSAPrivateKey_file() function adds the first private RSA key found to ssl.

The SSL_CTX_check_private_key() funciton checks the consistency of a private key with the corresponding certificate loaded into ctx. If more than one key/certificate pair (RSA/DSA) is installed, the last item installed will be checked. If, for example, the last item was an RSA certificate or key, the RSA key/certificate pair will be checked. TheSSL_check_private_key() function performs the same check for ssl. If no key/certificate was added for this ssl, the last item added into ctx will be checked.

NOTE The internal certificate store of OpenSSL can hold two private key/certificate pairs at a time: one key/certificate of type RSA and one key/certificate of type DSA. The certificate used depends on the cipher select. See *SSL_CTX_set_cipher_list*.

When reading certificates and private keys from file, files of type SSL_FILETYPE_ASN1 (also known as DER, binary encoding) can only contain one certificate or private key. Consequently, the SSL_CTX_use_certificate_chain_file() function is only applicable to PEM formatting. Files of type SSL_FILETYPE_PEM can contain more than one item.

The `SSL_CTX_use_certificate_chain_file()` function adds the first certificate found in the file to the certificate store. The other certificates are added to the store of chain certificates using the `SSL_CTX_add_extra_chain_cert()` function. There exists only one extra chain store, so that the same chain is appended to both types of certificates, RSA and DSA. If it is not intended to use both types of certificate at the same time, you should use the `SSL_CTX_use_certificate_chain_file()` function instead of the `SSL_CTX_use_certificate_file()` function. This allows the use of complete certificate chains even when no trusted CA storage is used or when the CA issuing the certificate shall not be added to the trusted CA storage.

If additional certificates are needed to complete the chain during the TLS negotiation, CA certificates are additionally looked up in the locations of trusted CA certificates. See *SSL_CTX_load_verify_locations*.

The private keys loaded from file can be encrypted. In order to successfully load encrypted keys, a function returning the passphrase must have been supplied,. See *SSL_CTX_set_default_passwd_cb*. (Certificate files also might be encrypted from the technical point of view, but it is unnecessary because the data in the certificate is considered public.)

RETURN VALUES

On success, the functions return 1. Otherwise, check the error stack to find the reason.

SEE ALSO

Functions: *ssl*, *SSL_new*, *SSL_clear*, *SSL_CTX_load_verify_locations*, *SSL_CTX_set_default_passwd_cb*, *SSL_CTX_set_cipher_list*, *SSL_CTX_add_extra_chain_cert*

SSL_do_handshake

NAME

SSL_do_handshake – Perform the initialization operations to setup an SSL connection.

SYNOPSIS

```
#include <openssl/ssl.h>
int SSL_do_handshake(
        SSL *s
);
```

DESCRIPTION

The `SSL_do_handshake()` function performs the initialization operations to set up an SSL connection. This API needs to be called before establishing an SSL connection.

RETURN VALUES

1

Indicates that the initialization for an SSL connection was completed successfully.

<=0

Indicates that the `SSL_do_handshake()` function failed to complete the initialization.

SSL_dup

NAME

SSL_dup – duplicates the specified SSL structure

SYNOPSIS

```
#include <openssl/ssl.h>
SSL *SSL_dup(
       SSL *s
);
```

DESCRIPTION

The `SSL_dup()` function duplicates the specified SSL structure and returns an address of the newly created SSL structure.

RETURN VALUES

The `SSL_dup()` function returns an address of the duplicated (new) SSL structure.

SEE ALSO

Functions: *SSL_dup_CA_list*

SSL_dup_CA_list

NAME

SSL_dup_CA_list – Duplicate the list of CAs

SYNOPSIS

```
#include <openssl/ssl.h>
STACK_OF(X509_NAME) *SSL_dup_CA_list(
       STACK_OF(X509_NAME) *sk
);
```

DESCRIPTION

The `SSL_dup_CA_list()` function duplicates the specified list of CAs and return an address of the newly created list of CAs.

RETURN VALUES

The `SSL_dup_CA_list()` function returns the following values:

STACK_OF (X509_NAMES)

List of CA names copied from the specified list of CA names.

NULL

This function failed to duplicate a list of CAs.

SEE ALSO

Functions: *SSL_dup*

SSL_free

NAME

SSL_free – Free an allocated SSL structure

SYNOPSIS

```
#include <openssl/ssl.h>
void SSL_free(
        SSL *ssl
);
```

DESCRIPTION

The `SSL_free()` function decrements the reference count of `ssl`, and removes the SSL structure pointed to by `ssl` and frees up the allocated memory if the the reference count has reached 0.

NOTES

The `SSL_free()` function also calls the freeing procedures for indirectly affected items, if applicable: the buffering BIO, the read and write BIOs, cipher lists specially created for this `ssl`, the `SSL_SESSION`. Do not explicitly free these indirectly freed up items before or after calling the `SSL_free()` function. Trying to free things twice may lead to program failure.

The ssl session has reference counts from two users: the SSL object, for which the reference count is removed by the `SSL_free()` function and the internal session cache. If the session is considered bad, because the `SSL_shutdown()` function was not called for the connection and the `SSL_set_shutdown()` function was not used to set the SSL_SENT_SHUTDOWN state. The session will also be removed from the session cache as required by RFC2246.

RETURN VALUES

The `SSL_free()` function does not provide diagnostic information.

SEE ALSO

Functions: *SSL_new*, *SSL_clear*, *SSL_shutdown*, *SSL_set_shutdown*, *ssl*

SSL_get_certificate

NAME

SSL_get_certificate – Return an X.509 certificate loaded in the SSL structure

SYNOPSIS

```
#include <openssl/ssl.h>
X509 *SSL_get_certificate(
        SSL *ssl
);
```

DESCRIPTION

The `SSL_get_certificate()` function returns an X.509 certificate loaded in the SSL structure. Before calling this function, an X509 certificate must be loaded into the SSL structure with another API (for example, `SSL_use_certificate_file()`, `SSL_CTX_use_certificate_file()`, etc.)

RETURN VALUES

The following return values can occur:

NULL

No X509 certificate is loaded in the SSL structure.

Pointer to an X509 structure

The return value points to an X509 certificate in the SSL structure.

SEE ALSO

Functions: *SSL_CTX_use_certificate*

SSL_get_ciphers

NAME

SSL_get_ciphers, SSL_get_cipher_list – Get list of available SSL_CIPHERs

SYNOPSIS

```
#include <openssl/ssl.h>

STACK_OF(SSL_CIPHER) *SSL_get_ciphers(
        SSL *ssl
);

const char *SSL_get_cipher_list(
         SSL *ssl, int priority
);
```

NOTES

The details of the ciphers obtained by the `SSL_get_ciphers()` function can be obtained using the `SSL_CIPHER_get_name()` family of functions.

Call the `SSL_get_cipher_list()` function with `priority` starting from 0 to obtain the sorted list of available ciphers, until NULL is returned.

RETURN VALUES

The `SSL_get_ciphers()` function returns the stack of available SSL_CIPHERs for `ssl`, sorted by preference. If `ssl` is NULL or no ciphers are available, NULL is returned.

The `SSL_get_cipher_list()` function returns a pointer to the name of the SSL_CIPHER listed for `ssl` with `priority`. If `ssl` is NULL, no ciphers are available, or there are less ciphers than `priority` available, NULL is returned.

SEE ALSO

Function: *ssl*, *SSL_CTX_set_cipher_list*, *SSL_CIPHER_get_name*

SSL_get_client_CA_list

NAME

SSL_get_client_CA_list, SSL_CTX_get_client_CA_list – Get list of client CAs

SYNOPSIS

```
#include <openssl/ssl.h>
STACK_OF(X509_NAME) *SSL_get_client_CA_list(
        SSL *s
);
STACK_OF (X509_NAME) *SSL_CTX_get_client_CA_list(
        SSL_CTX *ctx
);
```

DESCRIPTION

The `SSL_CTX_get_client_CA_list()` function returns the list of client CAs explicitly set for `ctx` using the `SSL_CTX_set_client_CA_list()` function.

The `SSL_get_client_CA_list()` function returns the list of client CAs explicitly set for `ssl` using the `SSL_set_client_CA_list()` function or `ssl`'s SSL_CTX object with the `SSL_CTX_set_client_CA_list()` function, when in server mode. In client mode, the `SSL_get_client_CA_list()` function returns the list of client CAs sent from the server, if any.

RETURN VALUES

The `SSL_CTX_set_client_CA_list()` and `SSL_set_client_CA_list()` functions do not return diagnostic information.

The `SSL_CTX_add_client_CA()` and `SSL_add_client_CA()` functions have the following return values:

STACK_OF (X509_NAMES)

List of CA names explicitly set (for **ctx** or in server mode) or send by the server (client mode).

NULL

No client CA list was explicitly set (for **ctx** or in server mode) or the server did not send a list of CAs (client mode).

SEE ALSO

Functions: *ssl*, *SSL_CTX_set_client_CA_list*

SSL_get_current_cipher

NAME

SSL_get_current_cipher, SSL_get_cipher, SSL_get_cipher_name, SSL_get_cipher_bits ,

SSL_get_cipher_version – Get SSL_CIPHER of a connection

SYNOPSIS

```
#include <openssl/ssl.h>

SSL_CIPHER *SSL_get_current_cipher(
       SSL *ssl
);

#define SSL_get_cipher(s) \
           SSL_CIPHER_get_name(SSL_get_current_cipher(s))
#define SSL_get_cipher_name(s) \
           SSL_CIPHER_get_name(SSL_get_current_cipher(s))
#define SSL_get_cipher_bits(s,np) \
           SSL_CIPHER_get_bits(SSL_get_current_cipher(s),np)
#define SSL_get_cipher_version(s) \
           SSL_CIPHER_get_version(SSL_get_current_cipher(s))
```

DESCRIPTION

The `SSL_get_current_cipher()` function returns a pointer to an SSL_CIPHER object containing the description of the actually used cipher of a connection established with the `ssl` object.

The `SSL_get_cipher()` and `SSL_get_cipher_name()` macros obtain the name of the currently used cipher. The `SSL_get_cipher_bits()` macro obtains the number of secret/algorithm bits used and the `SSL_get_cipher_version()` function returns the protocol name. See *SSL_CIPHER_get_name* for more details.

RETURN VALUES

The `SSL_get_current_cipher()` function returns the cipher actually used or NULL, when no session has been established.

SEE ALSO

Functions: *ssl*, *SSL_CIPHER_get_name*

SSL_get_default_timeout

NAME

SSL_get_default_timeout – Get default session timeout value

SYNOPSIS

```
#include <openssl/ssl.h>

long SSL_get_default_timeout(

        SSL *ssl

);
```

DESCRIPTION

The `SSL_get_default_timeout()` function returns the default timeout value assigned to SSL_SESSION objects negotiated for the protocol valid for ssl.

NOTES

Whenever a new session is negotiated, it is assigned a timeout value, after which it will not be accepted for session reuse. If the timeout value was not explicitly set using `SSL_CTX_set_timeout()`, the hardcoded default timeout for the protocol will be used.

The `SSL_get_default_timeout()` function returns this hardcoded value, which is 300 seconds for all currently supported protocols (SSLv2, SSLv3, and TLSv1).

SEE ALSO

Functions: *ssl*, *SSL_CTX_set_session_cache_mode*, *SSL_SESSION_get_time*, *SSL_CTX_flush_sessions*, *SSL_get_default_timeout*

SSL_get_error

NAME

SSL_get_error – Obtain result code for TLS/SSL I/O operation

SYNOPSIS

```
#include <openssl/ssl.h>
int SSL_get_error(
        SSL *ssl, int ret
);
```

DESCRIPTION

The `SSL_get_error()` function returns a result code (suitable for the C switch statement) for a preceding call to the `SSL_connect()`, `SSL_accept()`, `SSL_read()`, `SSL_peek()`, or `SSL_write()` functions on `ssl`. The value returned by that TLS/SSL I/O function must be passed to the `SSL_get_error()` function in parameter `ret`.

In addition to `ssl` and `ret`, the `SSL_get_error()` function inspects the current thread's OpenSSL error queue. Thus, the `SSL_get_error()` function must be used in the same thread that performed the TLS/SSL I/O operation, and no other OpenSSL function calls should appear in between. The current thread's error queue must be empty before the TLS/SSL I/O operation is attempted, or the `SSL_get_error()` function will not work reliably.

RETURN VALUES

The following return values can currently occur:

SSL_ERROR_NONE

The TLS/SSL I/O operation completed. This result code is returned if and only if `ret > 0`.

SSL_ERROR_ZERO_RETURN

The TLS/SSL connection has been closed. If the protocol version is SSL 3.0 or TLS 1.0, this result code is returned only if a closure alert has occurred in the protocol, i.e. if the connection has been closed cleanly. In this case SSL_ERROR_ZERO_RETURN does not necessarily indicate that the underlying transport has been closed.

SSL_ERROR_WANT_READ, SSL_ERROR_WANT_WRITE

The operation did not complete; the same TLS/SSL I/O function should be called again later. If, by then, the underlying BIO has data available for reading (if the result code is SSL_ERROR_WANT_READ) or allows writing data (SSL_ERROR_WANT_WRITE), then some TLS/SSL protocol progress will take place, i.e. at least part of an TLS/SSL record will be read or written. The retry may again lead to a SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE condition. There is no fixed upper limit for the number of iterations that may be necessary until progress becomes visible at application protocol level.

For socket BIOs (e.g. when the `SSL_set_fd()` function was used), the `select()` or `poll()` functions on the underlying socket can be used to determine when the TLS/SSL I/O function should be retried.

Caveat: Any TLS/SSL I/O function can lead to either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. In particular, the `SSL_read()` or `SSL_peek()` functions might write data, and the `SSL_write()` function might read data. This is because TLS/SSL handshakes can occur at any time during the protocol (initiated by either the client or the server); the `SSL_read()`, `SSL_peek()`, and `SSL_write()` functions will handle any pending handshakes.

SSL_ERROR_WANT_CONNECT

The operation did not complete; the same TLS/SSL I/O function should be called again later. The underlying BIO was not conected yet to the peer and the call would block in `connect()`. The SSL function should be called again when the connection is established. This message can only appear with a `BIO_s_connect()` BIO. In order to find out when the connection was successfully established on many platforms, the `select()` or `poll()` functions can be used for writing on the socket file descriptor.

SSL_ERROR_WANT_ X509_LOOKUP

The operation did not complete because an application callback set by the`SSL_CTX_set_client_cert_cb()` function has asked to be called again. The TLS/SSL I/O function should be called again later. Details depend on the application.

SSL_ERROR_SYSCALL

Some I/O error occurred. The OpenSSL error queue may contain more information on the error. If the error queue is empty (i.e. the `ERR_get_error()` functions returns 0), `ret` can be used to learn more about the error. If `ret == 0`, an EOF was observed that violates the protocol. If `ret == -1`, the underlying BIO reported an I/O error (for socket I/O on UNIX systems, consult `errno` for details).

SSL_ERROR_SSL

A failure in the SSL library occurred, usually a protocol error. The OpenSSL error queue contains more information on the error.

HISTORY

The `SSL_get_error()` function was added in SSLeay 0.8.

SEE ALSO

Functions: *ssl*, *err*

SSL_get_ex_data_X509_STORE_CTX_idx

NAME

SSL_get_ex_data_X509_STORE_CTX_idx – Get ex_data index to access SSL structure from X509_STORE_CTX

SYNOPSIS

```
#include <openssl/ssl.h>
int SSL_get_ex_data_X509_STORE_CTX_idx(
       void
);
```

DESCRIPTION

The `SSL_get_ex_data_X509_STORE_CTX_idx()` function returns the index number under which the pointer to the SSL object is stored into the X509_STORE_CTX object.

NOTES

Whenever an X509_STORE_CTX object is created for the verification of the peers certificate during a handshake, a pointer to the SSL object is stored into the X509_STORE_CTX object to identify the connection affected. To retrieve this pointer the `X509_STORE_CTX_get_ex_data()` function can be used with the correct index. This index is globally the same for all X509_STORE_CTX objects and can be retrieved using the `SSL_get_ex_data_X509_STORE_CTX_idx()` function. The index value is set when the `SSL_get_ex_data_X509_STORE_CTX_idx()` function is first called either by the application program directly or indirectly during other SSL setup functions or during the handshake.

The value depends on other index values defined for X509_STORE_CTX objects before the SSL index is created.

RETURN VALUES

>=0

The index value to access the pointer.

<0

An error occurred; check the error stack for a detailed error message.

EXAMPLES

The index returned from the `SSL_get_ex_data_X509_STORE_CTX_idx()` function allows you to access the SSL object for the connection to be accessed during the `verify_callback()` function when checking the peers certificate. Check the example in *SSL_CTX_set_verify*.

SEE ALSO

Functions: *ssl*, *SSL_CTX_set_verify*, *CRYPTO_set_ex_data*

SSL_get_ex_new_index

NAME

SSL_get_ex_new_index, SSL_set_ex_data, SSL_get_ex_data – Internal application specific data functions

SYNOPSIS

```
#include <openssl/ssl.h>

int SSL_get_ex_new_index(
        long argl, void *argp, CRYPTO_EX_new *new_func, CRYPTO_EX_dup *dup_func,
        CRYPTO_EX_free *free_func
);

int SSL_set_ex_data(
        SSL *ssl, int idx, void *arg
);

void *SSL_get_ex_data(
        SSL *ssl, int idx
);

typedef int new_func(
        void *parent, void *ptr, CRYPTO_EX_DATA *ad, int idx, long argl, void *argp
);

typedef void free_func(
        void *parent, void *ptr, CRYPTO_EX_DATA *ad, int idx, long argl, void *argp
);

typedef int dup_func(
        CRYPTO_EX_DATA *to, CRYPTO_EX_DATA *from, void *from_d, int idx, long argl, void
        *argp
);
```

DESCRIPTION

Several OpenSSL structures can have application specific data attached to them. These functions are used internally by OpenSSL to manipulate application specific data attached to a specific structure.

The `SSL_get_ex_new_index()` function is used to register a new index for application specific data.The `SSL_set_ex_data()` function is used to store application data at `arg` for `idx` into the `ssl` object.The `SSL_get_ex_data()` function is used to retrieve the information for `idx` from `ssl`.

A detailed description for the `*_get_ex_new_index()` functionality can be found in *RSA_get_ex_new_index*. The `*_get_ex_data()` and `*_set_ex_data()` functionality is described in *CRYPTO_set_ex_data*.

EXAMPLES

An example of the functionality is included in the `verify_callback()` function example in *SSL_CTX_set_verify*.

SSL_get_fd

NAME

SSL_get_fd – Get file descriptor linked to an SSL object

SYNOPSIS

```
#include <openssl/ssl.h>
 int SSL_get_fd(
        SSL *ssl
);
int SSL_get_rfd(
        SSL *ssl
);
int SSL_get_wfd(
        SSL *ssl
);
```

DESCRIPTION

The `SSL_get_fd()` function returns the file descriptor which is linked to `ssl`. The `SSL_get_rfd()` and `SSL_get_wfd()` functions return the file descriptors for the read or the write channel, which can be different. If the read and the write channel are different, the `SSL_get_fd()` function will return the file descriptor of the read channel.

RETURN VALUES

The following return values can occur:

-1

The operation failed, because the underlying BIO is not of the correct type (suitable for file descriptors).

>=0

The file descriptor linked to `ssl`.

SEE ALSO

Functions: *SSL_set_fd*, *ssl*, *bio*

SSL_get_finished

NAME

SSL_get_finished – Get the latest "Finished" message sent out and return the length of the message.

SYNOPSIS

```
#include <openssl/ssl.h>
size_t SSL_get_finished(
         SSL *s)
        (SSL *buf)
        (size_t count
);
```

DESCRIPTION

The `SSL_get_finished` function copies the latest "Finished" message (sent out from this side) to buf and returns the length of the "Finished" message of the SSL handshake.

RETURN VALUES

The `SSL_get_finished` function returns the length of the "Finished" message sent out from this side (client or server) during the SSL handshake.

SEE ALSO

Functions: *SSL_get_peer_finished*

SSL_get_info_callback

NAME

SSL_get_info_callback – Get the callback function set by SSL_set_info_callback

SYNOPSIS

```
#include <openssl/ssl.h>
void
       *SSL_get_info_callback SSL * ssl
);
```

DESCRIPTION

The `SSL_get_info_callback()` function returns a pointer to the informational callback function set (in info_callback of the SSL structure) by the`SSL_set_info_callback()` function.

RETURN VALUES

This function returns an address of the callback function set (in info_callback of the SSL structure) by the `SSL_set_info_callback()` function.

SEE ALSO

Functions: *SSL_set_info_callback*

SSL_get_peer_cert_chain

NAME

SSL_get_peer_cert_chain – Get the X509 certificate chain of the peer

SYNOPSIS

```
#include <openssl/ssl.h>
STACKOF(X509) *SSL_get_peer_cert_chain(
        SSL *ssl
);
```

DESCRIPTION

The `SSL_get_peer_cert_chain()` function returns a pointer to STACKOF(X509) certificates forming the certificate chain of the peer. If called on the client side, the stack also contains the peer's certificate; if called on the server side, the peer's certificate must be obtained separately using the `SSL_get_peer_certificate()` function. If the peer did not present a certificate, NULL is returned.

NOTES

The peer certificate chain is not necessarily available after reusing a session, in which case a NULL pointer is returned.

The reference count of the STACKOF(X509) object is not incremented. If the corresponding session is freed, the pointer must not be used.

RETURN VALUES

The following return values can occur:

NULL

No certificate was presented by the peer or no connection was established or the certificate chain is no longer available when a session is reused.

Pointer to a STACKOF(X509)

The return value points to the certificate chain presented by the peer.

SEE ALSO

Functions: *ssl*, *SSL_get_peer_certificate*

SSL_get_peer_certificate

NAME

SSL_get_peer_certificate – Get the X509 certificate of the peer

SYNOPSIS

```
#include <openssl/ssl.h>
X509 *SSL_get_peer_certificate(
        SSL *ssl
);
```

DESCRIPTION

The `SSL_get_peer_certificate()` function returns a pointer to the X509 certificate the peer presented. If the peer did not present a certificate, NULL is returned.

NOTES

A returned certificate does not indicate information about the verification state. Use the `SSL_get_verify_result()` function to check the verification state.

The reference count of the X509 object is incremented by one, so that it will not be destroyed when the session containing the peer certificate is freed. The X509 object must be explicitly freed using the `X509_free()` function.

RETURN VALUES

The following return values can occur:

NULL

No certificate was presented by the peer or no connection was established.

Pointer to an X509 certificate

The return value points to the certificate presented by the peer.

SEE ALSO

Functions: *ssl*, *SSL_get_verify_result*, *SSL_CTX_set_verify*

SSL_get_peer_finished

NAME

SSL_get_peer_finished – Gets the latest "Finished" message received and return the length of the message.

SYNOPSIS

```
#include <openssl/ssl.h>
size_t SSL_get_peer_finished(
         SSL *s)
        (void *buf)
        (size_t count
);
```

DESCRIPTION

The `SSL_get_finished()` function copies the latest "Finished" message (received on this side) to buf and returns the length of the "Finished" message of the SSL handshake.

RETURN VALUES

This function returns the length of the "Finished" message received on this side (client or server) during the SSL handshake.

SEE ALSO

Functions: *SSL_get_finished*

SSL_get_privatekey

NAME

SSL_get_privatekey – Get a private-key of the X.509 certificate loaded in the SSL structure

SYNOPSIS

```
#include <openssl/ssl.h>
EVP_PKEY *SSL_get_privatekey(
        SSL *s
);
```

DESCRIPTION

The `SSL_get_privatekey()` function returns a pointer to a private-key of the X.509 certificate loaded in the SSL structure. Before calling this function, a private-key must be loaded into the SSL structure with another API (for example, `SSL_use_PrivateKey_file()`, `SSL_CTX_use_PrivateKey_file()`, etc.)

RETURN VALUES

The following return values can occur:

NULL

No private-key is loaded in the SSL structure. Getting a private-key failed.

Pointer to an EVP_PKEY structure

The return value points to an EVP_PKEY structure in the SSL structure.

SEE ALSO

Functions: *SSL_use_PrivateKey*, *SSL_use_PrivateKey_ASN1*, *SSL_use_PrivateKey_file*, *SSL_use_RSAPrivateKey*, *SSL_use_RSAPrivateKey_ASN1*, *SSL_use_RSAPrivateKey_file*, *SSL_CTX_use_PrivateKey*, *SSL_CTX_use_PrivateKey_ASN1*, *SSL_CTX_use_PrivateKey_file*, *SSL_CTX_use_RSAPrivateKey*, *SSL_CTX_use_RSAPrivateKey_ASN1*, *SSL_CTX_use_RSAPrivateKey_file*

SSL_get_quiet_shutdown

NAME

SSL_get_quiet_shutdown – Get a value of the quiet-shutdown flag in the ssl data structure

SYNOPSIS

```
#include <openssl/ssl.h>
int SSL_get_quiet_shutdown(
        SSL *ssl
);
```

DESCRIPTION

The `SSL_get_quiet_shutdown()` function returns a mode of the quiet shutdown flag in the ssl structure.

RETURN VALUES

0

Indicates that the quiet-shutdown flag of the ssl structure is turned off.

1

Indicates that the quiet-shutdown flag of the ssl structure is turned on.

SEE ALSO

Functions: *SSL_set_quiet_shutdown*, *SSL_CTX_get_quiet_shutdown*, *SSL_CTX_set_quiet_shutdown*

SSL_get_rbio

NAME

SSL_get_rbio – Get BIO linked to an SSL object

SYNOPSIS

```
#include <openssl/ssl.h>
BIO *SSL_get_rbio(
        SSL *ssl
);
BIO *SSL_get_wbio(
        SSL *ssl
);
```

DESCRIPTION

The `SSL_get_rbio()` and `SSL_get_wbio()` functions return pointers to the BIOs for the read or the write channel, which can be different. The reference count of the BIO is not incremented.

RETURN VALUES

The following return values can occur:

NULL

No BIO was connected to the SSL object

Any other pointer

The BIO linked to `ssl`.

SEE ALSO

Functions: *SSL_set_bio*, *ssl*, *bio*

SSL_get_read_ahead

NAME

SSL_get_read_ahead – Get the read-ahead flag in the SSL structure.

SYNOPSIS

```
#include <openssl/ssl.h>
int SSL_get_read_ahead(
       SSL *w
);
```

DESCRIPTION

The `SSL_get_read_ahead()` function receives an SSL structure as an argument and returns the read-ahead flag in the SSL structure.

RETURN VALUES

The `SSL_get_read_ahead()` function returns a value of the read-ahead flag. Zero (the default value) indicates that the read-ahead flag is turned off.

SEE ALSO

Functions: *SSL_set_read_ahead*

SSL_get_session

NAME

SSL_get_session – Retrieve TLS/SSL session data

SYNOPSIS

```
#include <openssl/ssl.h>
SSL_SESSION *SSL_get_session(
        SSL *ssl
);
SSL_SESSION *SSL_get0_session(
        SSL *ssl
);
SSL_SESSION *SSL_get1_session(
        SSL *ssl
);
```

DESCRIPTION

The `SSL_get_session()` function returns a pointer to the SSL_SESSION actually used in `ssl`. The reference count of the SSL_SESSION is not incremented, so that the pointer can become invalid by other operations.

The`SSL_get0_session()` function is the same as the `SSL_get_session()` function.

The `SSL_get1_session()` function is the same as the `SSL_get_session()` function, but the reference count of the SSL_SESSION is incremented by one.

NOTES

The ssl session contains all information required to reestablish the connection without a new handshake.

The `SSL_get0_session()` function returns a pointer to the actual session. As the reference counter is not incremented, the pointer is only valid while the connection is in use. If the `SSL_clear()` function or the `SSL_free()` function is called, the session might be removed completely (if considered bad), and the pointer obtained will become invalid. Even if the session is valid, it can be removed at any time due to timeout during `SSL_CTX_flush_sessions()`.

If the data is to be kept, the `SSL_get1_session()` function will increment the reference count and the session will stay in memory until explicitly freed with `SSL_SESSION_free()`, regardless of its state.

RETURN VALUES

The following return values can occur:

NULL

There is no session available in **ssl**.

Pointer to an SSL

The return value points to the data of an SSL session.

SEE ALSO

Function: *ssl*, *SSL_free*, *SSL_clear*, *SSL_SESSION_free*

SSL_get_shared_ciphers

NAME

SSL_get_shared_ciphers – Get the shared ciphers from the SSL connection

SYNOPSIS

```
#include <openssl/ssl.h>

char *SSL_get_shared_ciphers(

        SSL *s
        char *buf,
        int len

);
```

DESCRIPTION

The `SSL_get_shared_ciphers()` function returns a pointer to a buffer (*buf) containing a list of shared ciphers.

The function can be used only for SSLv2 connection. It does not work for SSLv3 and TLSv1.

RETURN VALUES

The `SSL_get_shared_ciphers()` function returns a pointer to the buffer (*buf) containing a list of shared ciphers. This return value (pointer to characters) is the same as "char *buf", the second argument of this function.

SEE ALSO

Functions: *SSL_get_ciphers*

SSL_get_SSL_CTX

NAME

SSL_get_SSL_CTX – Return a pointer to the SSL_CTX object

SYNOPSIS

```
#include <openssl/ssl.h>
SSL_CTX *SSL_get_SSL_CTX(
        SSL *ssl
);
```

DESCRIPTION

The `SSL_get_SSL_CTX()` function returns a pointer to the SSL_CTX object, from which ssl was created with `SSL_new()`.

RETURN VALUES

The pointer to the SSL_CTX object is returned.

SEE ALSO

Functions: *ssl*, *SSL_new*

SSL_get_verify_result

NAME

SSL_get_verify_result – Get result of peer certificate verification

SYNOPSIS

```
#include <openssl/ssl.h>
long SSL_get_verify_result(
        SSL *ssl
);
```

DESCRIPTION

The `SSL_get_verify_result()` function returns the result of the verification of the X509 certificate presented by the peer, if any.

NOTES

The `SSL_get_verify_result()` function can only return one error code while the verification of a certificate can fail because of many reasons at the same time. Only the last verification error that occurred during the processing is available from `SSL_get_verify_result()`.

The verification result is part of the established session and is restored when a session is reused.

RESTRICTIONS

If no peer certificate was presented, the returned result code is X509_V_OK. This is because no verification error occurred; it does not indicate success. The `SSL_get_verify_result()` function is only useful in connection with the `SSL_get_peer_certificate()` function.

RETURN VALUES

The following return values can currently occur:

X509_V_OK

The verification succeeded or no peer certificate was presented.

Any other value

Documented in *verify*.

SEE ALSO

Commands: *verify*

Functions: *ssl*, *SSL_set_verify_result*, *SSL_get_peer_certificate*

SSL_get_version

NAME

SSL_get_version – Get the protocol version of a connection.

SYNOPSIS

```
#include <openssl/ssl.h>
const char *SSL_get_version(
        SSL *ssl
);
```

DESCRIPTION

The `SSL_get_cipher_version()` function returns the name of the protocol used for the connection `ssl`.

RETURN VALUES

The following strings can occur:

SSLv2

The connection uses the SSLv2 protocol.

SSLv3

The connection uses the SSLv3 protocol.

TLSv1

The connection uses the TLSv1 protocol.

unknown

This indicates that no version has been set (no connection established).

SEE ALSO

Functions: *ssl*

SSL_library_init

NAME

SSL_library_init, OpenSSL_add_ssl_algorithms, SSLeay_add_ssl_algorithms – Initialize SSL library by registering algorithms

SYNOPSIS

```
#include <openssl/ssl.h>

int SSL_library_init(
       void
);

#define OpenSSL_add_ssl_algorithms() SSL_library_init()
#define SSLeay_add_ssl_algorithms() SSL_library_init()
```

DESCRIPTION

The `SSL_library_init()` function registers the available ciphers and digests.

The `OpenSSL_add_ssl_algorithms()` and `SSLeay_add_ssl_algorithms()` functions are synonyms for the `SSL_library_init()` function.

NOTES

The `SSL_library_init()` function must be called before any other action takes place.

RESTRICTIONS

The `SSL_library_init()` function only registers ciphers. Another important initialization is the seeding of the PRNG (Pseudo Random Number Generator), which has to be performed separately.

RETURN VALUES

The `SSL_library_init()` function always returns 1, so it is safe to discard the return value.

EXAMPLES

A typical TLS/SSL application will start with the library initialization, will provide readable error messages and will seed the PRNG.

```
SSL_load_error_strings(); /* readable error messages */
SSL_library_init(); /* initialize library */
actions_to_seed_PRNG();
```

SEE ALSO

Functions: *ssl*, *SSL_load_error_strings*, *RAND_add*

SSL_load_client_CA_file

NAME

SSL_load_client_CA_file – Load certificate names from file

SYNOPSIS

```
#include <openssl/ssl.h>
STACK_OF(X509_NAME) *SSL_load_client_CA_file(
        const char *file
);
```

DESCRIPTION

The `SSL_load_client_CA_file()` function reads certificates from `file` and returns a STACK_OF(X509_NAME) with the subject names found.

NOTES

The `SSL_load_client_CA_file()` function reads a file of PEM formatted certificates and extracts the X509_NAMES of the certificates found. While the name suggests the specific usage as support for the `SSL_CTX_set_client_CA_list()` function, it is not limited to CA certificates.

EXAMPLES

Load names of CAs from file and use it as a client CA list:

```
SSL_CTX *ctx;
STACK_OF(X509_NAME) *cert_names;
...
cert_names = SSL_load_client_CA_file("/path/to/CAfile.pem");
if (cert_names != NULL)
  SSL_CTX_set_client_CA_list(ctx, cert_names);
else
  error_handling();
...
```

RETURN VALUES

The following return values can occur:

NULL

The operation failed, check out the error stack for the reason.

Pointer to STACK_OF (X509_NAME)

Pointer to the subject names of the successfully read certificates.

SEE ALSO

Functions: *ssl*, *SSL_CTX_set_client_CA_list*

SSL_new

NAME

SSL_new – Create a new SSL structure for a connection

SYNOPSIS

```
#include <openssl/ssl.h>
SSL *SSL_new(
        SSL_CTX *ctx
);
```

DESCRIPTION

The `SSL_new()` function creates a new `SSL` structure which is needed to hold the data for a TLS/SSL connection. The new structure inherits the settings of the underlying context `ctx`: connection method (SSLv2/v3/TLSv1), options, verification settings, timeout settings.

RETURN VALUES

The following return values can occur:

NULL

The creation of a new SSL structure failed. Check the error stack to find out the reason.

Pointer to an SSL structure

The return value points to an allocated SSL structure.

SEE ALSO

Functions: *SSL_free*, *SSL_clear*, *SSL_CTX_set_options*, *ssl*

SSL_peek

NAME

SSL_peek – Copy the data in the SSL buffer into the buffer passed to this API

SYNOPSIS

```
#include <openssl/ssl.h>
int SSL_peek(
        SSL * s)
        (void *buf)
        (int num
);
```

DESCRIPTION

The `SSL_peek()` function copies num bytes from the specified ssl into the buffer buf. In constrast to the `SSL_read()` function, the data in the SSL buffer is unmodified after the `SSL_peek()` operation.

RETURN VALUES

The following return values can occur:

>0

The peek operation was successful; the return value is the number of bytes actually copied from the TLS/SSL connection.

0

The peek operation was not successful; the SSL connection was closed by the peer by sending a ``close notify'' alert. The SSL_RECEIVED_SHUTDOWN flag in the ssl shutdown state is set. (See `SSL_shutdown()`, `SSL_set_shutdown()`. Call `SSL_get_error()` with the return value `ret` to determine whether an error occurred or the connection was shut down cleanly (SSL_ERROR_ZERO_RETURN).

SSLv2 (deprecated) does not support a shutdown alert protocol, so it can only be detected, whether the underlying connection was closed. It cannot be checked, whether the closure was initiated by the peer or by something else.

<0

The peek operation was not successful, because either an error occurred or action must be taken by the calling process. Call `SSL_get_error()` with the return value `ret` to find out the reason.

SEE ALSO

Functions: *SSL_read*

SSL_pending

NAME

SSL_pending – Obtain number of readable bytes buffered in an SSL object

SYNOPSIS

```
#include <openssl/ssl.h>
 int SSL_pending(
        SSL *ssl
);
```

DESCRIPTION

The `SSL_pending()` function returns the number of bytes which are available inside `ssl` for immediate read.

NOTES

Data are received in blocks from the peer. Therefore data can be buffered inside `ssl` and are ready for immediate retrieval with the `SSL_read()` function.

RESTRICTIONS

The `SSL_pending()` function takes into account only bytes from the TLS/SSL record that is being processed (if any). If the `SSL` object's read_ahead flag is set, additional protocol bytes may have been read containing more TLS/SSL records; these are ignored by the `SSL_pending()` function.

Up to OpenSSL 0.9.6, the `SSL_pending()` function does not check if the record type of pending data is application data.

RETURN VALUES

The number of bytes pending is returned.

SEE ALSO

Function: *SSL_read*, *ssl*

SSL_read

NAME

SSL_read – Read bytes from a TLS/SSL connection.

SYNOPSIS

```
#include <openssl/ssl.h>
int SSL_read(
        SSL *ssl, void *buf, int num
);
```

DESCRIPTION

The `SSL_read()` function tries to read `num` bytes from the specified `ssl` into the buffer `buf`.

NOTES

If necessary, the `SSL_read()` function will negotiate a TLS/SSL session, if not already explicitly performed by the `SSL_connect()` or `SSL_accept()` functions. If the peer requests a renegotiation, it will be performed transparently during the `SSL_read()` operation. The behavior of the `SSL_read()` function depends on the underlying BIO.

For the transparent negotiation to succeed, the `ssl` must have been initialized to client or server mode. This is not the case if a generic method is being used. (See *SSL_CTX_new*, so that the `SSL_set_connect_state()` or the `SSL_set_accept_state()` function must be used before the first call to an `SSL_read()` or `SSL_write()` function.)

The `SSL_read()` function is based on the SSL/TLS records. The data are received in records (with a maximum record size of 16kb for SSLv3/TLSv1). Only when a record has been completely received, can it be processed (decryption and check of integrity). Therefore, data that was not retrieved at the last call of `SSL_read()` can still be buffered inside the SSL layer and will be retired on the next call to `SSL_read()`. If `num` is higher than the number of bytes buffered, `SSL_read()` will return with the bytes buffered. If no more bytes are in the buffer, `SSL_read()` will trigger the processing of the next record. Only when the record has been received and processed completely will `SSL_read()` return reporting success. At most, the contents of the record will be returned. As the size of an SSL/TLS record may exceed the maximum packet size of the underlying transport, such as TCP, it may be necessary to read several packets from the transport layer before the record is complete and `SSL_read()` can succeed.

If the underlying BIO is `blocking`, the `SSL_read()` function will only return once the read operation has been finished or an error occurred, except when a renegotiation takes place, in which case a SSL_ERROR_WANT_READ may occur. This behavior can be controlled with the SSL_MODE_AUTO_RETRY flag of the `SSL_CTX_set_mode()` call.

If the underlying BIO is `non-blocking`, the `SSL_read()` function will also return when the underlying BIO could not satisfy the needs of `SSL_read()` to continue the operation. In this case a call to `SSL_get_error()` with the return value of `SSL_read()` will yield SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. As at any time a renegotiation is possible, a call to `SSL_read()` can also cause write operations. The calling process then must repeat the call after taking appropriate action to satisfy the needs of `SSL_read()`. The action depends on the underlying BIO. When using a non-blocking socket, nothing is to be done, but `select()` can be used to check for the required condition. When using a buffering BIO, such as a BIO pair, data must be written into or retrieved out of the BIO before being able to continue.

RESTRICTIONS

When an `SSL_read()` operation is repeated because of SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE, it must be repeated with the same arguments.

RETURN VALUES

The following return values can occur:

>0

The read operation was successful; the return value is the number of bytes actually read from the TLS/SSL connection.

0

The read operation was not successful, probably because no data was available. Call `SSL_get_error()` with the return value `ret` to find out whether an error occurred.

<0

The read operation was not successful, because either an error occurred or action must be taken by the calling process. Call `SSL_get_error()` with the return value `ret` to find the reason.

SEE ALSO

Functions: *SSL_get_error*, *SSL_write*, *SSL_CTX_set_mode*, *SSL_CTX_new*, *SSL_connect*, *SSL_accept* *SSL_set_connect_state*, *ssl*, *bio*

SSL_renegotiate

NAME

SSL_renegotiate – Turn on flags for renegotiation so that renegotiation will happen

SYNOPSIS

```
#include <openssl/ssl.h>
int SSL_renegotiate(
       SSL * s
);
```

DESCRIPTION

The `SSL_renegotiate()` function sets flags to initiate renegotiation. The renegotiation may happen at the next I/O operation provided that client/server are ready for renegotiation.

RETURN VALUES

1

Indicates that renegotiation flags have been set successfully.

0

Indicates that setting renegotiation flags failed.

SEE ALSO

SSL_CTX_sess_accept_renegotiate, *SSL_CTX_sess_connect_renegotiate*

SSL_rstate_string

NAME

SSL_rstate_string, SSL_rstate_string_long – Get textual description of state of an SSL object during read operation

SYNOPSIS

```
#include <openssl/ssl.h>
const char *SSL_rstate_string(
        SSL *ssl
);
const char *SSL_rstate_string_long(
        SSL *ssl
);
```

DESCRIPTION

The `SSL_rstate_string()` function returns a two letter string indicating the current read state of the SSL object ssl.

The `SSL_rstate_string_long()` function returns a string indicating the current read state of the SSL object ssl.

NOTES

When performing a read operation, the SSL/TLS engine must parse the record, consisting of header and body. When working in a blocking environment, `SSL_rstate_string[_long]()` should always return ``RD''/-read done.- This function is seldom needed.

RETURN VALUES

The `SSL_rstate_string()` and `SSL_rstate_string_long()` functions can return the following values:

RH (read header)

The header of the record is being evaluated.

RB (read body)

The body of the record is being evaluated.

RD (read done)

The record has been completely processed.

unknown

The read state is unknown. This should never happen.

SEE ALSO

Functions: *ssl*

SSL_SESSION_cmp

NAME

SSL_SESSION_cmp – Compare two SSL_SESSION structures

SYNOPSIS

```
#include <openssl/ssl.h>
int SSL_SESSION_cmp(
        SSL_SESSION *a)
        (SSL_SESSION *b
);
```

DESCRIPTION

`SSL_SESSION_cmp()` compares two SSL_SESSION structures. If the two structures are the same, this API return 0, otherwise non-zero value is returned.

RETURN VALUES

The following return values can occur:

0

SSL_SESSION structures, *a and *b, are the same.

non-zero value

SSL_SESSION structures, *a and *b, are different.

SEE ALSO

Functions: *SSL_get_session*, *SSL_SESSION_free*, *SSL_set_session*

SSL_SESSION_free

NAME

SSL_SESSION_free – Free an allocated SSL_SESSION structure

SYNOPSIS

```
#include <openssl/ssl.h>
void SSL_SESSION_free(
       SSL_SESSION *session
);
```

DESCRIPTION

The `SSL_SESSION_free()` function decrements the reference count of `session` and removes the `SSL_SESSION` structure pointed to by `session` and frees up the allocated memory, if the the reference count has reached 0.

RETURN VALUES

The `SSL_SESSION_free()` function does not provide diagnostic information.

SEE ALSO

Functions: *ssl*, *SSL_get_session*

SSL_SESSION_get_ex_new_index

NAME

SSL_SESSION_get_ex_new_index, SSL_SESSION_set_ex_data, SSL_SESSION_get_ex_data – Unternal application specific data functions

SYNOPSIS

```
#include <openssl/ssl.h>

 int SSL_SESSION_get_ex_new_index(

        long argl, void *argp, CRYPTO_EX_new *new_func, CRYPTO_EX_dup *dup_func,
        CRYPTO_EX_free *free_func

);

int SSL_SESSION_set_ex_data(

         SSL_SESSION *session, int idx, void *arg

);

void *SSL_SESSION_get_ex_data(

         SSL_SESSION *session, int idx

);

typedef int new_func(

        void *parent, void *ptr, CRYPTO_EX_DATA *ad, int idx, long argl, void *argp

);

typedef void free_func(

        oid *parent, void *ptr, CRYPTO_EX_DATA *ad, int idx, long argl, void *argp

);

typedef int dup_func(

        CRYPTO_EX_DATA *to, CRYPTO_EX_DATA *from, void *from_d, int idx, long argl, void
        *argp

);
```

DESCRIPTION

Several OpenSSL structures can have application specific data attached to them. These functions are used internally by OpenSSL to manipulate application specific data attached to a specific structure.

The `SSL_SESSION_get_ex_new_index()` function is used to register a new index for application specific data.

The `SSL_SESSION_set_ex_data()` function is used to store application data at `arg` for `idx` into the `session` object.

The `SSL_SESSION_get_ex_data()` function is used to retrieve the information for `idx` from `session`.

A detailed description for the `*_get_ex_new_index()` functionality can be found in `RSA_get_ex_new_index()`. The `*_get_ex_data()` and `*_set_ex_data()` functionality is described in `CRYPTO_set_ex_data()`.

RESTRICTIONS

The application data is only maintained for sessions held in memory. The application data is not included when dumping the session with the `i2d_SSL_SESSION()` function, as well as all functions indirectly calling the dump functions, such as the `PEM_write_SSL_SESSION()` and `PEM_write_bio_SSL_SESSION()`functions. It cannot be restored.

SEE ALSO

Functions: *ssl*, *RSA_get_ex_new_index*, *CRYPTO_set_ex_data*

SSL_SESSION_get_time

NAME

SSL_SESSION_get_time, SSL_SESSION_set_time, SSL_SESSION_get_timeout, SSL_SESSION_set_timeout – Retrieve and manipulate session time and timeout settings

SYNOPSIS

```
#include <openssl/ssl.h>
long SSL_SESSION_get_time(
        SSL_SESSION *s
);
long SSL_SESSION_set_time(
        SSL_SESSION *s, long tm
);
long SSL_SESSION_get_timeout(
        SSL_SESSION *s
);
long SSL_SESSION_set_timeout(
        SSL_SESSION *s, long tm
);
long SSL_get_time(
       SSL_SESSION *s
);
long SSL_set_time(
       SSL_SESSION *s, long tm
);
long SSL_get_timeout(
       SSL_SESSION *s
);
long SSL_set_timeout(
       SSL_SESSION *s, long tm
);
```

DESCRIPTION

The `SSL_SESSION_get_time()` function returns the time at which the session `s` was established. The time is given in seconds since the Epoch and therefore compatible to the time delivered by the `time()` call.

The `SSL_SESSION_set_time()` function replaces the creation time of the session `s` with the chosen value `tm`.

The `SSL_SESSION_get_timeout()` function returns the timeout value set for session `s` in seconds.

The `SSL_SESSION_set_timeout()` function sets the timeout value for session `s` in seconds to `tm`.

The `SSL_get_time()`, `SSL_set_time()`, `SSL_get_timeout()`, and `SSL_set_timeout()` functions are synonyms for the `SSL_SESSION_*()` counterparts.

NOTES

Sessions are expired by examining the creation time and the timeout value. Both are set at creation time of the session to the actual time and the default timeout value at creation, respectively, as set by the `SSL_CTX_set_timeout()` function. Using these functions it is possible to extend or shorten the lifetime of the session.

RETURN VALUES

The `SSL_SESSION_get_time()` and `SSL_SESSION_get_timeout()` functions return the currently valid values.

The `SSL_SESSION_set_time()` and `SSL_SESSION_set_timeout()` functions return 1 on success.

If any of the function is passed the NULL pointer for the session `s`, 0 is returned.

SEE ALSO

Functions: *ssl*, *SSL_CTX_set_timeout*

SSL_SESSION_hash

NAME

SSL_SESSION_hash – Return a session ID formatted as an unsigned long (32-bit)

SYNOPSIS

```
#include <openssl/ssl.h>
unsigned long SSL_SESSION_hash(
        SSL_SESSION *a
);
```

DESCRIPTION

The `SSL_SESSION_hash()` function formats session_id[0...3] of the received SSL_SESSION into a unsigned long data and returns it.

RETURN VALUES

unsigned long

session_id[0...3] formatted into unsigned long.

SEE ALSO

Functions: *SSL_get_session*, *SSL_SESSION_free*, *SSL_set_session*

SSL_SESSION_new

NAME

SSL_SESSION_new – Create a new SSL_SESSION structure

SYNOPSIS

```
#include <openssl/ssl.h>
SSL_SESSION *SSL_SESSION_new(
void
);
```

DESCRIPTION

The `SSL_SESSION_new()` function creates an SSL_SESSION structure and returns an address of the structure. 60*5+4 is set to timeout value of a newly created SSL_SESSION structure.

RETURN VALUES

Pointer to an SSL_SESSION structure

SEE ALSO

Functions: *SSL_SESSION_free*, *SSL_get_session*, *SSL_set_session*

SSL_SESSION_print

NAME

SSL_SESSION_print, SSL_SESSION_print_fp – Write data in the SSL_SESSION structure to the BIO or to an I/O stream specified by the file pointer

SYNOPSIS

```
#include <openssl/ssl.h>

int SSL_SESSION_print(

        BIO *bp
        SSL_SESSION *x

);

int SSL_SESSION_print_fp(

        FILE *fp
        SSL_SESSION *x

);
```

DESCRIPTION

The `SSL_SESSION_print()` writes SSL_SESSION information (including protocol type, cipher types, session id, and master key) into the BIO. If this function succeeds, it returns 1.

The `SSL_SESSION_print_fp()` writes SSL_SESSION information (including protocol type, cipher types, session id, and master key) into the FILE fp. If this function succeeds, it returns 1.

RETURN VALUES

Both `SSL_SESSION_print()` and `SSL_SESSION_print_fp()` functions return 1 on success and 0 on write errors.

SEE ALSO

Functions: *SSL_SESSION_free*, *SSL_get_session*, *SSL_set_session*

SSL_session_reused

NAME

SSL_session_reused – Query whether a reused session was negotiated during handshake

SYNOPSIS

```
#include <openssl/ssl.h>
int SSL_session_reused(
        SSL   *ssl
);
```

DESCRIPTION

The `SSL_session_reused()` function queries whether a reused session was negotiated during the handshake.

NOTES

During the negotiation, a client can propose to reuse a session. The server then looks up the session in its cache. If both client and server agree on the session, it will be reused and a flag is being set that can be queried by the application.

RETURN VALUES

The following return values can occur:

0

A new session was negotiated.

1

A session was reused.

SEE ALSO

Functions: *ssl*, *SSL_set_session*, *SSL_CTX_set_session_cache_mode*

SSL_set_bio

NAME

SSL_set_bio – Connect the SSL object with a BIO

SYNOPSIS

```
#include <openssl/ssl.h>
void SSL_set_bio(
        SSL *ssl, BIO *rbio, BIO *wbio
);
```

DESCRIPTION

The `SSL_set_bio()` function connects the BIOs `rbio` and `wbio` for the read and write operations of the TLS/SSL (encrypted) side of `ssl`.

The SSL engine inherits the behavior of `rbio` and `wbio`, respectively. If a BIO is non-blocking, the `ssl` will also have non-blocking behaviour.

If there was already a BIO connected to `ssl`, the `BIO_free()` function will be called (for both the reading and writing side, if different).

RETURN VALUES

The `SSL_set_bio()` function cannot fail.

SEE ALSO

Functions: *SSL_get_rbio*, *SSL_connect*, *SSL_accept*, *SSL_shutdown*, *ssl*, *bio*

SSL_set_connect_state

NAME

SSL_set_connect_state, SSL_get_accept_state – Prepare SSL object to work in client or server mode

SYNOPSIS

```
#include <openssl/ssl.h>
void SSL_set_connect_state(
        SSL *ssl
);
void SSL_set_accept_state(
        SSL *ssl
);
```

DESCRIPTION

The `SSL_set_connect_state()` function `ssl` to work in client mode.

The `SSL_set_accept_state()` function `ssl` to work in server mode.

NOTES

When the SSL_CTX object was created with `SSL_CTX_new()`, it was either assigned a dedicated client method, a dedicated server method, or a generic method, that can be used for both client and server connections. (The method might have been changed with `SSL_CTX_set_ssl_version()` or `SSL_set_ssl_method()`.)

In order to successfully accomplish the handshake, the SSL routines need to know whether they should act in server or client mode. If the generic method was used, this is not clear from the method itself and must be set with either `SSL_set_connect_state()` or `SSL_set_accept_state()`. If these routines are not called, the default value set when `SSL_new()` is called in server mode.

RETURN VALUES

The `SSL_set_connect_state()` and `SSL_set_accept_state()` functions do not return diagnostic information.

SEE ALSO

Functions: *ssl*, *SSL_new*, *SSL_CTX_new*, *SSL_CTX_set_ssl_version*

SSL_set_fd

NAME

SSL_set_fd – Connect the SSL object with a file descriptor

SYNOPSIS

```
#include <openssl/ssl.h>
int SSL_set_fd(
        SSL *ssl, int fd
);
int SSL_set_rfd(
        SSL *ssl, int fd
);
int SSL_set_wfd(
        SSL *ssl, int fd
);
```

DESCRIPTION

The `SSL_set_fd()` function sets the file descriptor `fd` as the input/output facility for the TLS/SSL (encrypted) side of `ssl`. The `fd` will typically be the socket file descriptor of a network connection.

When performing the operation, a `socket BIO` is automatically created to interface between the `ssl` and the `fd`. The BIO and the SSL engine inherit the behavior of `fd`. If `fd` is non-blocking, the `ssl` will also have non-blocking behavior.

If there was already a BIO connected to `ssl`, the `BIO_free()` function will be called (for both the reading and writing side, if different).

The `SSL_set_rfd()` and `SSL_set_wfd()` functions perform the respective action, but only for the read channel or the write channel, which can be set independently.

RETURN VALUES

The following return values can occur:

0

The operation failed. Check the error stack to find out why.

1

The operation succeeded.

SEE ALSO

Functions: *SSL_get_fd*, *SSL_set_bio*, *SSL_connect*, *SSL_accept*, *SSL_shutdown*, *ssl*, *bio*

SSL_set_info_callback

NAME

SSL_set_info_callback – Set a callback which will be called during the specified SSL connection

SYNOPSIS

```
#include <openssl/ssl.h>

void SSL_set_info_callback(

        SSL *ssl
        void *cb()

);
```

DESCRIPTION

The `SSL_set_info_callback()` function sets a callback which will be called during the specified SSL connection. This function is useful to trace an SSL connection.

SEE ALSO

Functions: *SSL_get_info_callback*

SSL_set_purpose

NAME

SSL_set_purpose – Set a purpose value to the SSL structure

SYNOPSIS

```
#include <openssl/ssl.h>
#include <openssl/x509v3.h> (to use the macros for purpose values)

int SSL_set_purpose(

        SSL *s
        int purpose

);
```

DESCRIPTION

The `SSL_set_purpose()` function sets a purpose value in the SSL structure. The purpose values and their macros are defined in x509v3.h as follows:

```
#define X509_PURPOSE_SSL_CLIENT 1
#define X509_PURPOSE_SSL_SERVER 2
#define X509_PURPOSE_NS_SSL_SERVER 3
#define X509_PURPOSE_SMIME_SIGN 4
#define X509_PURPOSE_SMIME_ENCRYPT 5
#define X509_PURPOSE_CRL_SIGN 6
#define X509_PURPOSE_ANY 7
```

The purpose value must be between 1 and 7. If an out-of-range value is passed, `SSL_set_purpose()` returns 0. Upon success, 1 is returned.

RETURN VALUES

The following return values can occur:

1

The purpose value was successfully set in the SSL structure.

0

Setting the purpose value in the SSL structure failed.

SEE ALSO

Functions: *SSL_CTX_set_purpose*

SSL_set_quiet_shutdown

NAME

SSL_set_quiet_shutdown – Set a value to the quiet-shutdown flag in the ssl data structure

SYNOPSIS

```
#include <openssl/ssl.h>

void SSL_set_quiet_shutdown(

        SSL *ssl
        int mode

);
```

DESCRIPTION

The `SSL_set_quiet_shutdown()` function sets a mode of quiet shutdown to the ssl structure. To turn on the quiet shutdown, mode == 1 needs to be passed. The mode == 0 turns off the quiet shutdown flag of the ssl structure. When `SSL_new()` creates an ssl structure, the value of the quiet-shutdown flag inherits from the quiet-shutdown flag in the SSL_CTX data structure.

SEE ALSO

Functions: *SSL_get_quiet_shutdown*, *SSL_CTX_get_quiet_shutdown*, *SSL_CTX_set_quiet_shutdown*

SSL_set_read_ahead

NAME

SSL_set_read_ahead – Sets the read-ahead flag in the SSL structure.

SYNOPSIS

```
#include <openssl/ssl.h>

void SSL_set_read_ahead(

       SSL *s
       int yes

);
```

DESCRIPTION

`SSL_get_read_ahead()` sets a value (int yes) to the read-ahead flag in the SSL structure. In order to turn on the flag, a nonzero value must be set.

SEE ALSO

Functions: *SSL_get_read_ahead*

SSL_set_session

NAME

SSL_set_session – Set a TLS/SSL session to be used during TLS/SSL connect

SYNOPSIS

```
#include <openssl/ssl.h>
int SSL_set_session(
        SSL *ssl, SSL_SESSION *session
);
```

DESCRIPTION

The `SSL_set_session()` function sets `session` to be used when the TLS/SSL connection is to be established. The `SSL_set_session()` function is only useful for TLS/SSL clients. When the session is set, the reference count of `session` is incremented by 1. If the session is not reused, the reference count is decremented again during `SSL_connect()`.

If there is already a session set inside `ssl` (because it was set with `SSL_set_session()` before or because the same `ssl` was already used for a connection), `SSL_SESSION_free()` will be called for that session.

RETURN VALUES

The following return values can occur:

0

The operation failed; check the error stack to find out the reason.

1

The operation succeeded.

SEE ALSO

Functions: *ssl*, *SSL_SESSION_free*, *SSL_CTX_set_session_cache_mode*

SSL_set_shutdown

NAME

SSL_set_shutdown, SSL_get_shutdown – Manipulate shutdown state of an SSL connection

SYNOPSIS

```
#include <openssl/ssl.h>
void SSL_set_shutdown(
        SSL *ssl, int mode
);
int SSL_get_shutdown(
        SSL *ssl
);
```

DESCRIPTION

The `SSL_set_shutdown()` function sets the shutdown state of `ssl` to `mode`. The `SSL_get_shutdown()` function returns the shutdown mode of `ssl`.

NOTES

The shutdown state of an ssl connection is a bitmask of:

0

No shutdown setting, yet.

SSL_SENT_SHUTDOWN

A `close notify` shutdown alert was sent to the peer, the connection is being considered closed and the session is closed and correct.

SSL_RECEIVED_SHUTDOWN

A shutdown alert was received form the peer, either a normal `close notify` or a fatal error.

SSL_SENT_SHUTDOWN and SSL_RECEIVED_SHUTDOWN can be set at the same time.

The shutdown state of the connection is used to determine the state of the ssl session. If the session is still open when `SSL_clear()` or `SSL_free()` is called, it is considered bad and removed according to RFC2246. The actual condition for a correctly closed session is SSL_SENT_SHUTDOWN. The `SSL_set_shutdown()` function can be used to set this state without sending a close alert to the peer (see *SSL_shutdown*).

If a `close notify` was received, SSL_RECEIVED_SHUTDOWN will be set. For setting SSL_SENT_SHUTDOWN, the application still must call `SSL_shutdown()` or `SSL_set_shutdown()`.

RETURN VALUES

The `SSL_set_shutdown()` function does not return diagnostic information. The `SSL_get_shutdown()` function returns the current setting.

SSL_set_trust

NAME

SSL_set_trust – sets a trust value to the SSL structure

SYNOPSIS

```
#include <openssl/ssl.h>
#include <openssl/x509.h> (to use the macros of trust values)

int SSL_set_trust(

        SSL *s
        int trust

);
```

DESCRIPTION

The `SSL_set_trust()` function sets a trust value in the SSL structure. The trust values and their macros are defined in x509v3.h as follows:

```
#define X509_TRUST_COMPAT 1
#define X509_TRUST_SSL_CLIENT 2
#define X509_TRUST_SSL_SERVER 3
#define X509_TRUST_EMAIL 4
#define X509_TRUST_OBJECT_SIGN 5
```

The trust value must be between 1 and 5. If an out-of-range value is passed, the `SSL_set_trust()` function returns 0. Upon success, 1 is returned.

RETURN VALUES

The following return values can occur:

1

The trust value was successfully set in the SSL structure.

0

Setting the trust value in the SSL structure failed.

SEE ALSO

Functions: *SSL_CTX_set_trust*

SSL_set_verify_result

NAME

SSL_set_verify_result – Override result of peer certificate verification

SYNOPSIS

```
#include <openssl/ssl.h>
void SSL_set_verify_result(
        SSL *ssl, long verify_result
);
```

DESCRIPTION

The `SSL_set_verify_result()` function sets `verify_result` of the object `ssl` to be the result of the verification of the X509 certificate presented by the peer, if any.

NOTES

The `SSL_set_verify_result()` function overrides the verification result. It only changes the verification result of the `ssl` object. It does not become part of the established session, so if the session is to be reused later, the original value will reappear.

The valid codes for `verify_result` are documented in *verify*.

RETURN VALUES

The `SSL_set_verify_result()` function does not provide a return value.

SEE ALSO

Commands: *verify*

Functions: *ssl*, *SSL_get_verify_result*, *SSL_get_peer_certificate*

SSL_shutdown

NAME

SSL_shutdown – Shut down a TLS/SSL connection

SYNOPSIS

```
#include <openssl/ssl.h>
int SSL_shutdown(
        SSL *ssl
);
```

DESCRIPTION

The `SSL_shutdown()` function shuts down an active TLS/SSL connection. It sends the `close notify` shutdown alert to the peer.

NOTES

The `SSL_shutdown()` function tries to send the `close notify` shutdown alert to the peer. Whether the operation succeeds or not, the SSL_SENT_SHUTDOWN flag is set and a currently open session is considered closed and good and will be kept in the session cache for further reuse.

The behavior of the `SSL_shutdown()` function depends on the underlying BIO.

If the underlying BIO is `blocking`, `SSL_shutdown()` will return only after the handshake finishes or an error occurs.

If the underlying BIO is `non-blocking`, `SSL_shutdown()` will also return when the underlying BIO could not satisfy the needs of `SSL_shutdown()` to continue the handshake. In this case, a call to `SSL_get_error()` with the return value of `SSL_shutdown()` will yield SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process then must repeat the call after taking appropriate action to satisfy the needs of `SSL_shutdown()`. The action depends on the underlying BIO. When using a non-blocking socket, nothing is to be done, but `select()` can be used to check for the required condition. When using a buffering BIO, like a BIO pair, data must be written into or retrieved out of the BIO before being able to continue.

RETURN VALUES

The following return values can occur:

1

The shutdown was successfully completed.

0

The shutdown was not successful. Call `SSL_get_error()` with the return value `ret` to find the reason.

-1

The shutdown was not successful because a fatal error occurred either at the protocol level or a connection failure occurred. It can also occur if action is needed to continue the operation for non-blocking BIOs. Call `SSL_get_error()` with the return value `ret` to find the reason.

SSL_state

NAME

SSL_state, SSL_state_string – Get a description of an SSL state

SYNOPSIS

```
#include <openssl/ssl.h>
int SSL_state(
        SSL *ssl
);
char *SSL_state_string(
        SSL *ssl
);
char *SSL_state_string_long(
        SSL *ssl
);
```

DESCRIPTION

These APIs are used to get the information about the current SSL state.

The `SSL_state()` function takes the address of the current SSL structure and return its state code (integer).

The `SSL_state_string()` function takes the address of the current SSL structure and return an address of its state code (string).

The `SSL_state_string_long()` function takes the address of the current SSL structure and return an address of its state (human-readable string).

RETURN VALUES

The `SSL_state()` function returns SSL status codes.

0x1000

Indicates SSL CONNECT state.

0x2000

Indicates SSL ACCEPT state.

0x0FFF

Indicates SSL MASK state.

0x1000|0x2000

Indicates SSL INIT state.

0x4000

Indicates SSL BEFORE state.

0x03

Indicates SSL OK state.

(0x04|(0x1000| 0x2000))

Indicates SSL RENEGOTIATE state.

The `SSL_state_string()` and `SSL_state_string_long()` functions return SSL state strings.

[Return values by `SSL_state_string()`] | [Return values by `SSL_state_string_long()`]

PINIT | "before SSL initialization"

Indicates that the SSL state is before SSL initialization.

AINIT | "before accept initialization"

Indicates that the SSL state is before accept initialization.

CINIT | "before connect initialization"

Indicates that the SSL state is before connect initialization.

SSLOK | "SSL negotiation finished successfully"

Indicates that SSL negotiation finished successfully.

Not Found | "SSL renegotiate ciphers"

Indicates that SSL is renegotiating ciphers.

Not Found | "before/connect initialization"

Indicates that the SSL state is before/connect initialization.

Not Found | "ok/connect SSL initialization"

Indicates ok/connect SSL initialization.

Not Found | "before/accept initialization"

Indicates that the SSL state is before/accept initialization.

Not Found | "ok/accept SSL initialization"

Indicates that the SSL state is ok/accept SSL initialization.

2CSENC | "SSLv2 client start encryption"

Indicates that SSLv2 is client starting encryption.

2SSENC | "SSLv2 server start encryption"

Indicates that SSLv2 server is starting encryption.

2SCH_A | "SSLv2 write client hello A"

Indicates that SSLv2 is writing client hello A.

2SCH_B | "SSLv2 write client hello B"

Indicates that SSLv2 is writing client hello B.

2GSH_A | "SSLv2 read server hello A"

Indicates that SSLv2 is reading server hello A.

2GSH_B | "SSLv2 read server hello B"

Indicates that SSLv2 is reading server hello B.

2SCMKA | "SSLv2 write client master key A"

Indicates that SSLv2 is writing client master key A.

2SCMKB | "SSLv2 write client master key B"

Indicates that SSLv2 is writing client master key B.

2SCF_A | "SSLv2 write client finished A"

Indicates that SSLv2 is writing client finished A.

2SCF_B | "SSLv2 write client finished B"

Indicates that SSLv2 is writing client finished B.

2SCC_A | "SSLv2 write client certificate A"

Indicates that SSLv2 is writing client certificate A.

2SCC_B | "SSLv2 write client certificate B"

Indicates that SSLv2 is writing client certificate B.

2SCC_C | "SSLv2 write client certificate C"

Indicates that SSLv2 is writing client certificate C.

2SCC_D | "SSLv2 write client certificate D"

Indicates that SSLv2 is writing client certificate D.

2GSV_A | "SSLv2 read server verify A"

Indicates that SSLv2 is reading server verify A.

2GSV_B | "SSLv2 read server verify B"

Indicates that SSLv2 is reading server verify B.

2GSF_A | "SSLv2 read server finished A"

Indicates that SSLv2 is reading server finished A.

2GSF_B | "SSLv2 read server finished B"

Indicates that SSLv2 is reading server finished B.

2GCH_A | "SSLv2 read client hello A"

Indicates that SSLv2 is reading client hello A.

2GCH_B | "SSLv2 read client hello B"

Indicates that SSLv2 is reading client hello B.

2GCH_C | "SSLv2 read client hello C"

Indicates that SSLv2 is reading client hello C.

2SSH_A | "SSLv2 write server hello A"

Indicates that SSLv2 is writing server hello A.

2SSH_B | "SSLv2 write server hello B"

Indicates that SSLv2 is writing server hello B.

2GCMKA | "SSLv2 read client master key A"

Indicates that SSLv2 is reading client master key A.

2GCMKB | "SSLv2 read client master key B"

Indicates that SSLv2 is reading client master key B.

2SSV_A | "SSLv2 write server verify A"

Indicates that SSLv2 is writing server verify A.

2SSV_B | "SSLv2 write server verify B"

Indicates that SSLv2 is writing server verify B.

2SSV_C | "SSLv2 write server verify C"

Indicates that SSLv2 is writing server verify C.

2GCF_A | "SSLv2 read client finished A"

Indicates that SSLv2 is reading client finished A.

2GCF_B | "SSLv2 read client finished B"

Indicates that SSLv2 is reading client finished B.

2SSF_A | "SSLv2 write server finished A"

Indicates that SSLv2 is writing server finished A.

2SSF_B | "SSLv2 write server finished B"

Indicates that SSLv2 is writing server finished B.

2SRC_A | "SSLv2 write request certificate A"

Indicates that SSLv2 is writing request certificate A.

2SRC_B | "SSLv2 write request certificate B"

Indicates that SSLv2 is writing request certificate B.

2SRC_C | "SSLv2 write request certificate C"

Indicates that SSLv2 is writing request certificate C.

2SRC_D | "SSLv2 write request certificate D"

Indicates that SSLv2 is writing request certificate D.

2X9GSC | "SSLv2 X509 read server certificate"

Indicates that SSLv2 X509 is reading server certificate.

2X9GCC | "SSLv2 X509 read client certificate"

Indicates that SSLv2 X509 is reading client certificate.

3FLUSH | "SSLv3 flush data"

Indicates that SSLv3 is flushing data.

3WCH_A | "SSLv3 write client hello A"

Indicates that SSLv3 is writing client hello A.

3WCH_B | "SSLv3 write client hello B"

Indicates that SSLv3 is writing client hello B.

3RSH_A | "SSLv3 read server hello A"

Indicates that SSLv3 is reading server hello A.

3RSH_B | "SSLv3 read server hello B"

Indicates that SSLv3 is reading server hello B.

3RSC_A | "SSLv3 read server certificate A"

Indicates that SSLv3 is reading server certificate A.

3RSC_B | "SSLv3 read server certificate B"

Indicates that SSLv3 is reading server certificate B.

3RSKEA | "SSLv3 read server key exchange A"

Indicates that SSLv3 is reading server key exchange A.

3RSKEB | "SSLv3 read server key exchange B"

Indicates that SSLv3 is reading server key exchange B.

3RCR_A | "SSLv3 read server certificate request A"

Indicates that SSLv3 is reading server certificate request A.

3RCR_B | "SSLv3 read server certificate request B"

Indicates that SSLv3 is reading server certificate request B.

3RSD_A | "SSLv3 read server done A"

Indicates that SSLv3 is reading server done A.

3RSD_B | "SSLv3 read server done B"

Indicates that SSLv3 is reading server done B.

3WCC_A | "SSLv3 write client certificate A"

Indicates that SSLv3 is writing client certificate A.

3WCC_B | "SSLv3 write client certificate B"

Indicates that SSLv3 is writing client certificate B.

3WCC_C | "SSLv3 write client certificate C"

Indicates that SSLv3 is writing client certificate C.

3WCC_D | "SSLv3 write client certificate D"

Indicates that SSLv3 is writing client certificate D.

3WCKEA | "SSLv3 write client key exchange A"

Indicates that SSLv3 is writing client key exchange A.

3WCKEB | "SSLv3 write client key exchange B"

Indicates that SSLv3 is writing client key exchange B.

3WCV_A | "SSLv3 write certificate verify A"

Indicates that SSLv3 is writing certificate verify A.

3WCV_B | "SSLv3 write certificate verify B"

Indicates that SSLv3 is writing certificate verify B.

3WCCSA | "SSLv3 write change cipher spec A"

Indicates that SSLv3 is writing change cipher spec A.

3WCCSB | "SSLv3 write change cipher spec B"

Indicates that SSLv3 is writing change cipher spec B.

3WFINA | "SSLv3 write finished A"

Indicates that SSLv3 is writing finished A.

3WFINB | "SSLv3 write finished B"

Indicates that SSLv3 is writing finished B.

3RCCSA | "SSLv3 read change cipher spec A"

Indicates that SSLv3 is reading change cipher spec A.

3RCCSB | "SSLv3 read change cipher spec B"

Indicates that SSLv3 is reading change cipher spec B.

3RFINA | "SSLv3 read finished A"

Indicates that SSLv3 is reading finished A.

3RFINB |"SSLv3 read finished B"

Indicates that SSLv3 is reading finished B.

3WHR_A | "SSLv3 write hello request A"

Indicates that SSLv3 is writing hello request A.

3WHR_B | "SSLv3 write hello request B"

Indicates that SSLv3 is writing hello request B.

3WHR_C | "SSLv3 write hello request C"

Indicates that SSLv3 is writing hello request C.

3RCH_A | "SSLv3 read client hello A"

Indicates that SSLv3 is reading client hello A.

3RCH_B | "SSLv3 read client hello B"

Indicates that SSLv3 is reading client hello B.

3RCH_C | "SSLv3 read client hello C"

Indicates that SSLv3 is reading client hello C.

3WSH_A | "SSLv3 write server hello A"

Indicates that SSLv3 is writing server hello A.

3WSH_B | "SSLv3 write server hello B"

Indicates that SSLv3 is writing server hello B.

3WSC_A | "SSLv3 write certificate A"

Indicates that SSLv3 is writing certificate A.

3WSC_B | "SSLv3 write certificate B"

Indicates that SSLv3 is writing certificate B.

3WSKEA | "SSLv3 write key exchange A"

Indicates that SSLv3 is writing key exchange A.

3WSKEB | "SSLv3 write key exchange B"

Indicates that SSLv3 is writing key exchange B.

3WCR_A | "SSLv3 write certificate request A"

Indicates that SSLv3 is writing certificate request A.

3WCR_B | "SSLv3 write certificate request B"

Indicates that SSLv3 is writing certificate request B.

3WSD_A | "SSLv3 write server done A"

Indicates that SSLv3 is writing server done A.

3WSD_B | "SSLv3 write server done B"

Indicates that SSLv3 is writing server done B.

3RCC_A | "SSLv3 read client certificate A"

Indicates that SSLv3 is reading client certificate A.

3RCC_B | "SSLv3 read client certificate B"

Indicates that SSLv3 is reading client certificate B.

3RCKEA | "SSLv3 read client key exchange A"

Indicates that SSLv3 is reading client key exchange A.

3RCKEB | "SSLv3 read client key exchange B"

Indicates that SSLv3 is reading client key exchange B.

3RCV_A | "SSLv3 read certificate verify A"

Indicates that SSLv3 is reading certificate verify A.

3RCV_B | "SSLv3 read certificate verify B"

Indicates that SSLv3 is reading certificate verify B.

23WCHA | "SSLv2/v3 write client hello A"

Indicates that SSLv2/v3 is writing client hello A.

23WCHB | "SSLv2/v3 write client hello B"

Indicates that SSLv2/v3 is writing client hello B.

23RSHA | "SSLv2/v3 read server hello A"

Indicates that SSLv2/v3 is reading server hello A.

23RSHB | "SSLv2/v3 read server hello B"

Indicates that SSLv2/v3 is reading server hello B.

23RCHA | "SSLv2/v3 read client hello A"

Indicates that SSLv2/v3 is reading client hello A.

23RCHB | "SSLv2/v3 read client hello B"

Indicates that SSLv2/v3 is reading client hello B.

UNKWN | "unknown state"

Indicates that the SSL state is unknown.

SSL_state_string

NAME

SSL_state_string, SSL_state_string_long – Get textual description of state of an SSL object

SYNOPSIS

```
#include <openssl/ssl.h>
const char *SSL_state_string(
        SSL *ssl)
const char *SSL_state_string_long(
        SSL *ssl
);
```

DESCRIPTION

The `SSL_state_string()` function returns a 6 letter string indicating the current state of the SSL object ssl.

The `SSL_state_string_long()` function returns a string indicating the current state of the SSL object ssl.

NOTES

During its use, an SSL objects passes several states. The state is internally maintained. Querying the state information is not very informative before or when a connection has been established. However, it can be of significant interest during the handshake.

When using non-blocking sockets, the function call performing the handshake might return an SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE condition, so that the `SSL_state_string[_long]()` function might be called.

For both blocking or non-blocking sockets, the details state information can be used within the `info_callback()` function set with the `SSL_set_info_callback()` call.

RETURN VALUES

Detailed description of possible states to be included later.

SEE ALSO

Functions: *ssl*, *SSL_CTX_set_info_callback*

SSL_version

NAME

SSL_version – Get a version of the SSL structure

SYNOPSIS

```
#include <openssl/ssl.h>
int SSL_version(
       SSL *s
);
```

DESCRIPTION

The `SSL_version()` function returns an SSL version (one of SSL2_VERSION, SSL3_VERSION, TLS1_VERSION).

RETURN VALUES

SSL2_VERSION

Indicates that the SSL version is SSLv2.

SSL3_VERSION

Indicates that the SSL version is SSLv3.

TLS1_VERSION

Indicates that the SSL version is TLSv1.

SSL_want

NAME

SSL_want, SSL_want_nothing, SSL_want_read, SSL_want_write, SSL_want_x509_lookup – Obtain state information TLS/SSL I/O operation

SYNOPSIS

```
#include <openssl/ssl.h>
int SSL_want(
        SSL  *ssl
);
int SSL_want_nothing(
        SSL  *ssl
);
int SSL_want_read(
        SSL  *ssl
);
int SSL_want_write(
        SSL  *ssl
);
int SSL_want_x509_lookup(
         SSL *ssl
);
```

DESCRIPTION

The `SSL_want()` function returns state information for the SSL object ssl.

The other `SSL_want_*()` functions are shortcuts for the possible states returned by `SSL_want()`.

NOTES

The `SSL_want()` function examines the internal state information of the SSL object. Its return values are similar to that of `SSL_get_error()`. Unlike `SSL_get_error()`, which also evaluates the error queue, the results are obtained by examining an internal state flag only. Therefore, the information must only be used for normal operation under non-blocking I/O. Error conditions are not handled and must be treated using `SSL_get_error()`.

The result returned by the `SSL_want()` function should always be consistent with the result of the `SSL_get_error()` function.

RETURN VALUES

The following return values can occur for `SSL_want()`:

SSL_NOTHING

There is no data to be written or to be read.

SSL_WRITING

There are data in the SSL buffer that must be written to the underlying BIO layer in order to complete the actual `SSL_*()` operation. A call to `SSL_get_error()` should return SSL_ERROR_WANT_WRITE.

SSL_READING

More data must be read from the underlying BIO layer in order to complete the actual `SSL_*()` operation. A call to `SSL_get_error()` should return SSL_ERROR_WANT_READ.

SSL_X509_LOOKUP

The operation did not complete because an application callback set by `SSL_CTX_set_client_cert_cb()` has asked to be called again. A call to `SSL_get_error()` should return SSL_ERROR_WANT_X509_LOOKUP.

The `SSL_want_nothing()`, `SSL_want_read()`, `SSL_want_write()`, and `SSL_want_x509_lookup()` functions return 1 when the corresponding condition is true or 0 otherwise.

SEE ALSO

Functions: *ssl*, *err*, *SSL_get_error*

SSL_write

NAME

SSL_write – Write bytes to a TLS/SSL connection.

SYNOPSIS

```
#include <openssl/ssl.h>
int SSL_write(
        SSL *ssl, const void *buf, int num
);
```

DESCRIPTION

The `SSL_write()` function writes `num` bytes from the buffer `buf` into the specified `ssl` connection.

NOTES

If necessary, the `SSL_write()` function will negotiate a TLS/SSL session, if not already explicitly performed by `SSL_connect()` or `SSL_accept()`. If the peer requests a renegotiation, it will be performed transparently during the `SSL_write()` operation. The behavior of `SSL_write()` depends on the underlying BIO.

For the transparent negotiation to succeed, the `ssl` must have been initialized to client or server mode. This is not the case if a generic method is being used (see *SSL_CTX_new*), so that `SSL_set_connect_state()` or `SSL_set_accept_state()` must be used before the first call to an `SSL_read()` or `SSL_write()` function.

If the underlying BIO is `blocking`, `SSL_write()` will only return once the write operation finishes or an error occurs, except when a renegotiation take place, in which case an SSL_ERROR_WANT_READ might occur. This behavior can be controlled with the SSL_MODE_AUTO_RETRY flag of the `SSL_CTX_set_mode()` function.

If the underlying BIO is `non-blocking`, `SSL_write()` will also return, when the underlying BIO could not satisfy the needs of `SSL_write()` to continue the operation. In this case a call to `SSL_get_error()` with the return value of `SSL_write()` will yield SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. As at any time a renegotiation is possible, a call to `SSL_write()` can also cause read operations. The calling process then must repeat the call after taking appropriate action to satisfy the needs of `SSL_write()`. The action depends on the underlying BIO. When using a non-blocking socket, nothing is to be done, but `select()` can be used to check for the required condition. When using a buffering BIO, like a BIO pair, data must be written into or retrieved out of the BIO before being able to continue.

The `SSL_write()` function will only return with success when the complete contents of `buf` of length `num` has been written. This default behavior can be changed with the SSL_MODE_ENABLE_PARTIAL_WRITE option of `SSL_CTX_set_mode()`. When this flag is set, `SSL_write()` will also return with success when a partial write successfully completes. In this case, the `SSL_write()` operation is considered complete. The bytes are sent and a new `SSL_write()` operation with a new buffer (with the previously sent bytes removed) must be started. A partial write is performed with the size of a message block, which is 16kB for SSLv3/TLSv1.

RESTRICTIONS

When an `SSL_write()` operation has to be repeated because of SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE, it must be repeated with the same arguments.

RETURN VALUES

The following return values can occur:

>0

The write operation was successful, the return value is the number of bytes actually written to the TLS/SSL connection.

0

The write operation was not successful. Call `SSL_get_error()` with the return value `ret` to find out, whether an error occurred.

<0

The write operation was not successful, because either an error occurred or action must be taken by the calling process. Call `SSL_get_error()` with the return value `ret` to find the reason.

SEE ALSO

Functions: *SSL_get_error*, *SSL_read*, *SSL_CTX_set_mode*, *SSL_CTX_new*, *SSL_connect*, *SSL_accept* *SSL_set_connect_state*, *ssl*, *bio*

threads

NAME

threads: CRYPTO_set_locking_callback, CRYPTO_set_id_callbackCRYPTO_num_locks, CRYPTO_set_dynlock_create_callback, CRYPTO_set_dynlock_lock_callback, CRYPTO_set_dynlock_destroy_callback, CRYPTO_get_new_dynlockid, CRYPTO_destroy_dynlockid, CRYPTO_lock – OpenSSL thread support

SYNOPSIS

```
#include <openssl/crypto.h>

void CRYPTO_set_locking_callback(
        void *locking_functionint  mode, int n, const char *file , int line
);

void CRYPTO_set_id_callback(
        unsigned long *id_functionvoid
);

int CRYPTO_num_locks(
       void
);

/* struct CRYPTO_dynlock_value needs to be defined by the user */
 struct CRYPTO_dynlock_value;

void CRYPTO_set_dynlock_create_callback(
       struct CRYPTO_dynlock_value * *dyn_create_function char *file, int line
);

void CRYPTO_set_dynlock_lock_callback(
       void *dyn_lock_functionint  mode, struct CRYPTO_dynlock_value *l, const char
       *file, int line
);

void CRYPTO_set_dynlock_destroy_callback(
       void *dyn_destroy_function struct CRYPTO_dynlock_value *l, const char * file, int
       line
);

int CRYPTO_get_new_dynlockid(
        void
);

void CRYPTO_destroy_dynlockid(
        int i
);

void CRYPTO_lock(
       int  mode)
       (int n)
       (const char *file)
```

```
            (int  line
      );
#define CRYPTO_w_lock(type)\
CRYPTO_lock(CRYPTO_LOCK|CRYPTO_WRITE,type,__FILE__,__LINE__)

#define CRYPTO_w_unlock(type)\
CRYPTO_lock(CRYPTO_UNLOCK|CRYPTO_WRITE,type,__FILE__,__LINE__)

#define CRYPTO_r_lock(type)\
CRYPTO_lock(CRYPTO_LOCK|CRYPTO_READ,type,__FILE__,__LINE__)

#define CRYPTO_r_unlock(type)\
CRYPTO_lock(CRYPTO_UNLOCK|CRYPTO_READ,type,__FILE__,__LINE__)

#define CRYPTO_add(addr,amount,type)\
CRYPTO_add_lock(addr,amount,type,__FILE__,__LINE__)
```

DESCRIPTION

OpenSSL can safely be used in multi-threaded applications provided that at least two callback functions are set.

The `locking_function(int mode, int n, const char *file, int line)` is needed to perform locking on shared data structures. Multi-threaded applications will crash at random if it is not set.

The `locking_function()` (int mode, int n, const char *file, int line) is needed to perform locking on shared data structures. (Note that OpenSSL uses a number of global data structures that will be implicitly shared whenever multiple threads use OpenSSL.) Multi-threaded applications will crash at random if it is not set.

The `file` and `line` are the file number of the function setting the lock. They can be useful for debugging.

The `id_function(void)` function returns a thread ID. It is not needed on Windows nor on platforms where `getpid()` returns a different ID for each thread (most notably Linux).

Additionally, OpenSSL supports dynamic locks, and some parts of OpenSSL need it for better performance. To enable this, the following is required:

- Three additional callback functions: `dyn_create_function`, `dyn_lock_function` and `dyn_destroy_function`.
- A structure defined with the data that each lock needs to handle.

The struct `CRYPTO_dynlock_value` has to be defined to contain whatever structure is needed to handle locks.

The `dyn_create_function(const char *file, int line)` is needed to create a lock. Multi-threaded applications might crash at random if it is not set.

The `dyn_lock_function(int mode, CRYPTO_dynlock *l, const char *file, int line)` is needed to perform locking off dynamic lock numbered `n`. Multi-threaded applications might crash at random if it is not set.

The `dyn_destroy_function(CRYPTO_dynlock *l, const char *file, int line)` is needed to destroy the lock `l`. Multi-threaded applications might crash at random if it is not set.

The `CRYPTO_get_new_dynlockid()` function is used to create locks. It will call `dyn_create_function` for the actual creation.

The `CRYPTO_destroy_dynlockid()` function is used to destroy locks. It will call `dyn_destroy_function` for the actual destruction.

The `CRYPTO_lock()` function is used to lock and unlock the locks. The mode is a bitfield describing what should be done with the lock. The value of `n` is the number of the lock as returned from the `CRYPTO_get_new_dynlockid()` function. The mode can be combined from the following values. These values are pairwise exclusive, with undefined behavior if misused. For example, CRYPTO_READ and CRYPTO_WRITE should not be used together:

```
CRYPTO_LOCK0x01
CRYPTO_UNLOCK0x02
CRYPTO_READ0x04
CRYPTO_WRITE0x08
```

RETURN VALUES

The `CRYPTO_num_locks()` function returns the required number of locks.

`CRYPTO_get_new_dynlockid()` function returns the index to the newly created lock.

The other functions return no values.

NOTES

You can determine if OpenSSL was configured with thread support:

```
#define OPENSSL_THREAD_DEFINES
#include <openssl/opensslconf.h>
#if defined(THREADS)
   // thread support enabled
#else
   // no thread support
#endif
```

Also, dynamic locks are not used internally by OpenSSL.

HISTORY

The `CRYPTO_set_locking_callback()` and `CRYPTO_set_id_callback()` functions are available in all versions of SSLeay and OpenSSL. The `CRYPTO_num_locks()` function was added in OpenSSL 0.9.4. All functions dealing with dynamic locks were added in OpenSSL 0.9.5b-dev.

SEE ALSO

Functions: *crypto*

verify

NAME

verify – Utility to verify certificates

SYNOPSIS

`openssl verify [-CApath` *`directory`*`] [-CAfile` *`filename`*`] [-purpose` *`purpose`*`] [-untrusted` *`filename`*`] [-help] [-issuer_checks] [-verbose] [-] [-certificates]`

OPTIONS

`CApath` *`directory`*

A directory of trusted certificates. The certificates should have names of the form `hash.0` or have symbolic links to them of this form. Under UNIX the `c_rehash` script will automatically create symbolic links to a directory of certificates. (Hash is the hashed certificate subject name. See the `hash` option of the `x509` command.)

`CAfile` *`filename`*

A file of trusted certificates. The file should contain multiple certificates in `PEM` format concatenated together.

`untrusted` *`filename`*

A file of untrusted certificates. The file should contain multiple certificates

`purpose` *`purpose`*

The intended use for the certificate. Without this option no chain verification will be done. Currently accepted uses are `sslclient`, `sslserver`, `nssslserver`, `smimesign`, `smimeencrypt`. See the Verify Operation section for more information.

`help`

Prints out a usage message.

`verbose`

Prints extra information about the operations being performed.

`issuer_checks`

Prints out diagnostics relating to searches for the issuer certificate of the current certificate. This shows why each candidate issuer certificate was rejected. However the presence of rejection messages does not itself imply that anything is wrong. During the normal verify process, several rejections may take place.

`-`

Marks the last option. All arguments following this are assumed to be certificate files. This is useful if the first certificate filename begins with a `-`.

`certificates`

One or more certificates to verify. If no certificate filenames are included then an attempt is made to read a certificate from standard input. They should all be in `PEM` format.

DESCRIPTION

The `verify` utility verifies certificate chains. It uses the same functions as the internal SSL and S/MIME verification. However, there is one crucial difference between the verify operations performed by the `verify` program. Wherever possible an attempt is made to continue after an error. Usually the verify operation would halt on the first error. This allows all the problems with a certificate chain to be determined.

The verify operation consists of a number of separate steps.

First, a certificate chain is built, starting from the supplied certificate and ending in the root CA. It is an error if the whole chain cannot be built. The chain is built by looking up the issuer's certificate of the current certificate. If a certificate is found which is its own issuer it is assumed to be the root CA.

The process of looking up the issuers certificate involves a number of steps. In versions of OpenSSL before 0.9.5a the first certificate whose subject name matched the issuer of the current certificate was assumed to be the issuer's certificate. In OpenSSL 0.9.6 and later all certificates whose subject name matches the issuer name of the current certificate are subject to further tests. The relevant authority key identifier components of the current certificate (if present) must match the subject key identifier (if present) and issuer and serial number of the candidate issuer. In addition, the keyUsage extension of the candidate issuer (if present) must permit certificate signing.

1. The lookup first looks in the list of untrusted certificates and if no match is found the remaining lookups are from the trusted certificates. The root CA is always looked up in the trusted certificate list. If the certificate to verify is a root certificate then an exact match must be found in the trusted list.
2. The second operation is to check every untrusted certificate's extensions for consistency with the supplied purpose. If the `purpose` option is not included then no checks are done. The supplied or leaf certificate must have extensions compatible with the supplied purpose and all other certificates must also be valid CA certificates. The precise extensions required are described in more detail in the Certificate Extensions section of the `x509` utility.
3. The third operation is to check the trust settings on the root CA. The root CA should be trusted for the supplied purpose. For compatibility with previous versions of SSLeay and OpenSSL a certificate with no trust settings is considered to be valid for all purposes.
4. The final operation is to check the validity of the certificate chain. The validity period is checked against the current system time and the `notBefore` and `notAfter` dates in the certificate. The certificate signatures are also checked at this point.

If all operations complete successfully then the certificate is considered valid. If any operation fails then the certificate is not valid.

RESTRICTIONS

Although the issuer checks are a considerably improvement over the old technique they still suffer from limitations in the underlying `X509_LOOKUP` API. One consequence of this is that trusted certificates with matching subject name must either appear in a file (as specified by the `CAfile` option) or a directory (as specified by `CApath`. If they occur in both then only the certificates in the file will be recognized.

Previous versions of OpenSSL assume certificates with matching subject name are identical and mishandled them.

ERRORS

When a verify operation fails, the output messages can be somewhat cryptic. The general form of the error message is:

```
server.pem: /C=AU/ST=Queensland/O=CryptSoft Pty Ltd/CN=Test CA (1024 bit)
 error 24 at 1 depth lookup:invalid CA certificate
```

The first line contains the name of the certificate being verified followed by the subject name of the certificate. The second line contains the error number and the depth. The depth is the number of the certificate being verified when a problem was detected, starting with zero for the certificate being verified itself then 1 for the CA that signed the certificate and so on. Finally a text version of the error number is presented.

A list of the error codes and messages is shown below. This also includes the name of the error code as defined in the header file `x509_vfy.h` Some of the error codes are defined but never returned. These are described as unused.

`0 X509_V_OK: ok`

The operation was successful.

`2 X509_V_ERR_UNABLE_TO_GET_ISSUER_CERT: unable to get issuer certificate`

The issuer certificate could not be found. This occurs if the issuer certificate of an untrusted certificate cannot be found.

`3 X509_V_ERR_UNABLE_TO_GET_CRL unable to get certificate CRL`

The CRL of a certificate could not be found. Unused.

`4 X509_V_ERR_UNABLE_TO_DECRYPT_CERT_SIGNATURE: unable to decrypt certificate's signature`

The certificate signature could not be decrypted. This means that the actual signature value could not be determined rather than it not matching the expected value, this is only meaningful for RSA keys.

`5 X509_V_ERR_UNABLE_TO_DECRYPT_CRL_SIGNATURE: unable to decrypt CRL's signature`

The CRL signature could not be decrypted. This means that the actual signature value could not be determined rather than it not matching the expected value. Unused.

`6 X509_V_ERR_UNABLE_TO_DECODE_ISSUER_PUBLIC_KEY: unable to decode issuer public key`

The public key in the certificate `SubjectPublicKeyInfo` could not be read.

`7 X509_V_ERR_CERT_SIGNATURE_FAILURE: certificate signature failure`

The signature of the certificate is invalid.

`8 X509_V_ERR_CRL_SIGNATURE_FAILURE: CRL signature failure`

The signature of the certificate is invalid. Unused.

`9 X509_V_ERR_CERT_NOT_YET_VALID: certificate is not yet valid`

The certificate is not yet valid. The `notBefore` date is after the current time.

`10 X509_V_ERR_CERT_HAS_EXPIRED: certificate has expired`

The certificate has expired. The `notAfter` date is before the current time.

`10 X509_V_ERR_CRL_NOT_YET_VALID: CRL is not yet valid`

The CRL is not yet valid. Unused.

`11 X509_V_ERR_CERT_HAS_EXPIRED: Certificate has expired`

The certificate has expired. The `notAfter` date is before the current time.

`11 X509_V_ERR_CRL_NOT_YET_VALID: CRL is not yet valid`

The CRL is not yet valid. Unused.

12 X509_V_ERR_CRL_HAS_EXPIRED: CRL has expired

The CRL has expired. Unused.

13 X509_V_ERR_ERROR_IN_CERT_NOT_BEFORE_FIELD: format error in certificate's notBefore field

The certificate notBefore field contains an invalid time.

14 X509_V_ERR_ERROR_IN_CERT_NOT_AFTER_FIELD: format error in certificate's notAfter field

The certificate notAfter field contains an invalid time.

15 X509_V_ERR_ERROR_IN_CRL_LAST_UPDATE_FIELD: format error in CRL's lastUpdate field

The CRL lastUpdate field contains an invalid time. Unused.

16 X509_V_ERR_ERROR_IN_CRL_NEXT_UPDATE_FIELD: format error in CRL's nextUpdate field

The CRL nextUpdate field contains an invalid time. Unused.

17 X509_V_ERR_OUT_OF_MEM: out of memory

An error occurred trying to allocate memory. This should never happen.

18 X509_V_ERR_DEPTH_ZERO_SELF_SIGNED_CERT: self signed certificate

The passed certificate is self signed and the same certificate cannot be found in the list of trusted certificates.

19 X509_V_ERR_SELF_SIGNED_CERT_IN_CHAIN: self signed certificate in certificate chain

The certificate chain could be built up using the untrusted certificates but the root could not be found locally.

20 X509_V_ERR_UNABLE_TO_GET_ISSUER_CERT_LOCALLY: unable to get local issuer certificate

The issuer certificate of a locally looked up certificate could not be found. This normally means the list of trusted certificates is not complete.

21 X509_V_ERR_UNABLE_TO_VERIFY_LEAF_SIGNATURE: unable to verify the first certificate

No signatures could be verified because the chain contains only one certificate and it is not self signed.

22 X509_V_ERR_CERT_CHAIN_TOO_LONG: certificate chain too long

The certificate chain length is greater than the supplied maximum depth. Unused.

23 X509_V_ERR_CERT_REVOKED: certificate revoked

The certificate has been revoked. Unused.

24 X509_V_ERR_INVALID_CA: invalid CA certificate

A CA certificate is invalid. Either it is not a CA or its extensions are not consistent with the supplied purpose.

25 X509_V_ERR_PATH_LENGTH_EXCEEDED: path length constraint exceeded

The basicConstraints pathlength parameter has been exceeded.

26 X509_V_ERR_INVALID_PURPOSE: unsupported certificate purpose

The supplied certificate cannot be used for the specified purpose.

27 X509_V_ERR_CERT_UNTRUSTED: certificate not trusted

The root CA is not marked as trusted for the specified purpose.

`28 X509_V_ERR_CERT_REJECTED: certificate rejected`

The root CA is marked to reject the specified purpose.

`29 X509_V_ERR_SUBJECT_ISSUER_MISMATCH: subject issuer mismatch`

The current candidate issuer certificate was rejected because its subject name did not match the issuer name of the current certificate. This is only displayed when the `issuer_checks` option is set.

`30 X509_V_ERR_AKID_SKID_MISMATCH: authority and subject key identifier mismatch`

The current candidate issuer certificate was rejected because its subject key identifier was present and did not match the authority key identifier current certificate. This is only displayed when the `issuer_checks` option is set.

`31 X509_V_ERR_AKID_ISSUER_SERIAL_MISMATCH: authority and issuer serial number mismatch`

The current candidate issuer certificate was rejected because its issuer name and serial number was present and did not match the authority key identifier of the current certificate. This is only displayed when the `issuer_checks` option is set.

`32 X509_V_ERR_KEYUSAGE_NO_CERTSIGN: key usage does not include certificate signing`

The current candidate issuer certificate was rejected because its keyUsage extension does not permit certificate signing.

`50 X509_V_ERR_APPLICATION_VERIFICATION: application verification failure`

An application specific error. Unused.

SEE ALSO

Commands: *x509*

version

NAME

version – Prints OpenSSL version information

SYNOPSIS

`openssl version [-a] [-b] [-f] [-o] [-p] [-v]`

OPTIONS

`a`

All information. This is the same as setting all the other flags.

`b`

The date the current version of OpenSSL was built.

`f`

Compilation flags.

`o`

Option information. Various options set when the library was built.

`p`

Platform setting.

`v`

The current OpenSSL version.

DESCRIPTION

The `version` command prints version information about OpenSSL.

NOTES

The output of `openssl version a` would typically be used when sending a bug report.

x509

NAME

x509 – Certificate display and signing utility

SYNOPSIS

```
openssl x509 [-inform DER|PEM|NET] [-outform DER|PEM|NET] [-keyform DER|PEM]
[-CAform DER|PEM] [-CAkeyform DER|PEM] [-in filename] [-out filename] [-serial]
[-hash] [-subject] [-issuer] [-nameopt option] [-email] [-startdate] [-enddate]
[-purpose] [-dates] [-modulus] [-fingerprint] [-alias] [-noout] [-trustout]
[-clrtrust] [-clrreject] [-addtrust arg] [-addreject arg] [setalias arg] [days arg]
[-signkey filename] [-x509toreq] [-req] [-CA filename] [-CAkey filename]
[-CAcreateserial] [-CAserial filename] [-text] [-C] [-md2 | md5 | sha1 | mdc2]
[-clrext] [-extfile filename] [-extensions section]
```

INPUT, OUTPUT AND GENERAL PURPOSE OPTIONS

inform DER|PEM|NET

Specifies the input format. Normally the command will expect an X509 certificate but this can change if other options such as `req` are present. The DER format is the DER encoding of the certificate, and PEM is the base64 encoding of the DER encoding with header and footer lines added. The NET option is an obscure Netscape server format that is obsolete.

outform DER|PEM|NET

Specifies the output format. The options have the same meaning as the `inform` option.

in *filename*

Specifies the input filename to read a certificate from or standard input if this option is not specified.

out *filename*

Specifies the output filename to write to or standard output by default.

md2|md5|sha1|mdc2

The digest to use. This affects any signing or display option that uses a message digest, such as the `fingerprint`, `signkey` and `CA` options. If not specified then MD5 is used. If the key being used to sign with is a DSA key then this option has no effect. SHA1 is always used with DSA keys.

DISPLAY OPTIONS

-text

Prints out the certificate in text form. Full details are output including the public key, signature algorithms, issuer and subject names, serial number any extensions present and any trust settings.

-noout

Prevents output of the encoded version of the request.

-modulus

Prints out the value of the modulus of the public key contained in the certificate.

`-serial`

Outputs the certificate serial number.

`-hash`

Outputs the hash of the certificate subject name. This is used in OpenSSL to form an index to allow certificates in a directory to be looked up by subject name.

`-subject`

Outputs the subject name.

`-issuer`

Outputs the issuer name.

`-nameopt option`

Option which determines how the subject or issuer names are displayed. This option may be used more than once to set multiple options. See the Name Options section for more information.

`-email`

Outputs the email address if any.

`-startdate`

Prints the start date of the certificate, that is the notBefore date.

`-enddate`

Prints the expiration date of the certificate, that is the notAfter date.

`-dates`

Prints the start and expiration dates of a certificate.

`-fingerprint`

Prints the digest of the DER encoded version of the whole certificate.

`-C`

Outputs the certificate in the form of a C source file.

The `alias` and `purpose` options are display options but are described in the Trust Options section.

TRUST OPTIONS

These options are experimental and may change.

A `trusted certificate` is an ordinary certificate which has several additional pieces of information attached to it, such as the permitted and prohibited uses of the certificate and an alias.

Usually when a certificate is being verified, at least one certificate must be trusted. By default a trusted certificate must be stored locally and must be a root CA. Any certificate chain ending in this CA is then usable for any purpose.

Trust settings are only used with a root CA. They allow finer control over the purposes of the root CA. For example, a CA may be trusted for SSL client but not SSL server use.

See the description of the `verify` utility for more information on the meaning of trust settings.

Future versions of OpenSSL will recognize trust settings on any certificate, not just root CAs.

`trustout`

Causes `x509` to output a trusted certificate. An ordinary or trusted certificate can be input, but by default an ordinary certificate is output and any trust settings are discarded. With the `trustout` option a trusted certificate is output. A trusted certificate is automatically output if any trust settings are modified.

`setalias` *arg*

Sets the alias of the certificate. This will allow the certificate to be referred to using a nickname, such as `Steve's Certificate`.

`alias`

Outputs the certificate alias, if any.

`clrtrust`

Clears all the permitted or trusted uses of the certificate.

`clrreject`

Clears all the prohibited or rejected uses of the certificate.

`addtrust` *arg*

Adds a trusted certificate use. Any object name can be used here but only `clientAuth` (SSL client use), `serverAuth` (SSL server use) and `emailProtection` (S/MIME email) are used. Other OpenSSL applications may define additional uses.

`addreject` *arg*

Adds a prohibited use. It accepts the same values as the `addtrust` option.

`purpose`

Performs tests on the certificate extensions and outputs the results.

Certificate Extensions

The `purpose` option checks the certificate extensions and determines what the certificate can be used for. The actual checks are complex and include various hacks and workarounds to handle broken certificates and software.

The same code is used when verifying untrusted certificates in chains.

The basicConstraints extension CA flag is used to determine whether the certificate can be used as a CA. If the CA flag is true then it is a CA. If the CA flag is false then it is not a CA. All CAs should have the CA flag set to true.

If the basicConstraints extension is absent then the certificate is considered to be a possible CA. Other extensions are checked according to the intended use of the certificate. A warning is given in this case because the certificate should not be regarded as a CA. However, it is allowed to be a CA to work around some broken software.

If the certificate is a V1 certificate (and thus has no extensions) and it is self signed it is also assumed to be a CA, but a warning is given. This is to work around the problem of Verisign roots which are V1 self signed certificates.

If the keyUsage extension is present then additional restraints are made on the uses of the certificate. A CA certificate must have the keyCertSign bit set if the keyUsage extension is present.

The extended key usage extension places additional restrictions on the certificate uses. If this extension is present (whether critical or not) the key can only be used for the purposes specified.

The comments about basicConstraints and keyUsage and V1 certificates apply to all CA certificates. A complete description of each test follows:

- `SSL Client`

  The extended key usage extension must be absent or include the web client authentication OID. The keyUsage extension must be absent or it must have the digitalSignature bit set. Netscape certificate type must be absent or it must have the SSL client bit set.

- `SSL Client CA`

  The extended key usage extension must be absent or include the web client authentication OID. Netscape certificate type must be absent or it must have the SSL CA bit set. This is used as a work around if the basicConstraints extension is absent.

- `SSL Server`

  The extended key usage extension must be absent or include the web server authentication and/or one of the SGC OIDs. The keyUsage extension must be absent or it must have the digitalSignature, the keyEncipherment set or both bits set. Netscape certificate type must be absent or have the SSL server bit set.

- `SSL Server CA`

  The extended key usage extension must be absent or include the web server authentication and/or one of the SGC OIDs. Netscape certificate type must be absent or the SSL CA bit must be set. This is used as a work around if the basicConstraints extension is absent.

- `Netscape SSL Server`

  For Netscape SSL clients to connect to an SSL server it must have the keyEncipherment bit set if the keyUsage extension is present. This isn't always valid because some cipher suites use the key for digital signing. Otherwise it is the same as a normal SSL server.

- `Common S/MIME Client Tests`

  The extended key usage extension must be absent or include the email protection OID. Netscape certificate type must be absent or should have the S/MIME bit set. If the S/MIME bit is not set in netscape certificate type then the SSL client bit is tolerated as an alternative but a warning is shown. This is because some Verisign certificates don't set the S/MIME bit.

- `S/MIME Signing`

  In addition to the common S/MIME client tests the digitalSignature bit must be set if the keyUsage extension is present.

- `S/MIME Encryption`

  In addition to the common S/MIME tests the keyEncipherment bit must be set if the keyUsage extension is present.

- `S/MIME CA`

  The extended key usage extension must be absent or include the email protection OID. Netscape certificate type must be absent or must have the S/MIME CA bit set. This is used as a work around if the basicConstraints extension is absent.

- `CRL Signing`

  The keyUsage extension must be absent or it must have the CRL signing bit set.

- `CRL Signing CA`

  The normal CA tests apply, except in this case the basicConstraints extension must be present.

SIGNING OPTIONS

The `x509` command can be used to sign certificates and requests. It can thus behave like a mini CA.

`signkey` *`filename`*

Causes the input file to be self-signed using the supplied private key. If the input file is a certificate it sets the issuer name to the subject name (i.e. It makes it self-signed, changes the public key to the supplied value, and changes the start and end dates.) The start date is set to the current time and the end date is set to a value determined by the `days` option. Any certificate extensions are retained unless the `clrext` option is supplied. If the input is a certificate request, then a self-signed certificate is created using the supplied private key using the subject name in the request.

`clrext`

Deletes any extensions from a certificate. This option is used when a certificate is being created from another certificate, such as with the `signkey` or the `CA` options. Normally all extensions are retained.

`keyform PEM|DER`

Specifies the format (`DER` or `PEM`) of the private key file used in the `signkey` option.

`days` *`arg`*

Specifies the number of days to make a certificate valid. The default is 30 days.

`x509toreq`

Converts a certificate into a certificate request. The `signkey` option is used to pass the required private key.

`req`

By default a certificate is expected on input. With this option, a certificate request is expected instead.

`CA` *`filename`*

Specifies the CA certificate to be used for signing. When this option is present `x509` behaves like a mini CA. The input file is signed by this CA using this option, meaning its issuer name is set to the subject name of the CA and it is digitally signed using the CA's private key.

This option is normally combined with the `req` option. Without the `req` option the input is a certificate which must be self-signed.

`CAkey` *`filename`*

Sets the CA private key to sign a certificate with. If this option is not specified then it is assumed that the CA private key is present in the CA certificate file.

`CAserial` *`filename`*

Sets the CA serial number file to use.

When the CA option is used to sign a certificate it uses a serial number specified in a file. This file consist of one line containing an even number of hex digits with the serial number to use. After each use the serial number is incremented and written out to the file again.

The default filename consists of the CA certificate file base name with .srl appended. For example, if the CA certificate file is called mycacert.pem it expects to find a serial number file called mycacert.srl.

CAcreateserial *filename*

Creates a CA serial number file if it does not exist. It will contain the serial number 02 and the certificate being signed will have the number 1 as its serial number. Normally, if the CA option is specified and the serial number file does not exist it is an error.

extfile *filename*

File containing certificate extensions to use. If not specified then no extensions are added to the certificate.

extensions *section*

The section to add certificate extensions from. If this option is not specified then the extensions should either be contained in the unnamed (default) section or the default section should contain a variable called extensions which contains the section to use.

NAME OPTIONS

The nameopt command line option determines how the subject and issuer names are displayed. If no nameopt option is present the default oneline format is used which is compatible with previous versions of OpenSSL. Each option is described in detail below. All options can be preceded by a - to turn the option off. Usually, only the first four are used.

compat

Uses the old format. This is equivalent to specifying no name options.

RFC2253

Displays names compatible with RFC2253 equivalent to esc_2253, esc_ctrl, esc_msb, utf8, dump_nostr, dump_unknown, dump_der, sep_comma_plus, dn_rev and sname.

oneline

A oneline format which is more readable than RFC2253. It is equivalent to esc_2253, esc_ctrl, esc_msb, utf8, dump_nostr, dump_der, use_quote, sep_comma_plus_spc, spc_eq and sname options.

multiline

A multiline format. It is equivalent to esc_ctrl, esc_msb, sep_multiline, spc_eq and lname.

esc_2253

Escapes the special characters required by RFC2253 in a field. These characters are ,+"<>;. Additionally # is escaped at the begining of a string and a space character at the beginning or end of a string.

esc_ctrl

Escapes control characters. These characters are those with ASCII values less than 0x20 (space) and the delete (0x7f) character. They are escaped using the RFC2253 \XX notation, where *XX* are two hex digits representing the character value.

esc_msb

Escapes characters with the MSB set, that is with ASCII values larger than 127.

use_quote

Escapes some characters by surrounding the whole string with " characters. Without the option, all escaping is done with the \ character.

utf8

Converts all strings to UTF8 format first. This is required by RFC2253. If you have a UTF8 compatible terminal then the use of this option (and not setting esc_msb) may result in the correct display of multibyte (international) characters. Is this option is not present, then multibyte characters larger than 0xff will be represented using the format \UXXXX for 16 bits and \WXXXXXXXX for 32 bits. Also, if this option is off any UTF8Strings will be converted to their character form first.

no_type

Does not attempt to interpret multibyte characters in any way. Their content octets are merely dumped as though one octet represents each character. This is useful for diagnostic purposes but will result in rather odd looking output.

show_type

Shows the type of the ASN1 character string. The type precedes the field contents. For example, BMPSTRING: Hello World.

dump_der

When this option is set any fields that need to be hexdumped will be dumped using the DER encoding of the field. Otherwise, just the content octets will be displayed. Both options use the RFC2253 #XXXX... format.

dump_nostr

Dumps non-character string types, such as OCTET STRING. If this option is not set then non-character string types will be displayed as though each content octet repesents a single character.

dump_all

Dumps all fields. This option when used with dump_der allows the DER encoding of the structure to be unambiguously determined.

dump_unknown

Dumps any field whose OID is not recognized by OpenSSL.

sep_comma_plus, sep_comma_plus_space, sep_semi_plus_space, sep_multiline

Determines the field separators. The first character is between RDNs and the second between multiple AVAs. (Multiple AVAs are very rare and their use is discouraged.) The options ending in space additionally place a space after the separator to make it more readable. The sep_multiline uses a linefeed character for the RDN separator and a spaced + for the AVA separator. It also indents the fields by four characters.

dn_rev

Reverses the fields of the DN. This is required by RFC2253. As a side effect this also reverses the order of multiple AVAs, but this is permissible.

`nofname`, `sname`, `lname`, `oid`

Alter how the field name is displayed. The `nofname` option does not display the field at all. The `sname` option uses the short name form (CN for commonName, for example), and the `lname` option uses the long form. The `oid` option represents the OID in numerical form and is useful for diagnostic purpose.

`spc_eq`

Places spaces around the = character which follows the field name.

DESCRIPTION

The `x509` command is a multipurpose certificate utility. It can be used to display certificate information, convert certificates to various forms, sign certificate requests such as a mini CA, or edit certificate trust settings. Since there are a large number of options they are divided into various sections.

NOTES

The `PEM` format uses the following header and footer lines:

```
-----BEGIN CERTIFICATE----
-----END CERTIFICATE----
```

It will also handle files containing:

```
-----BEGIN X509 CERTIFICATE----
-----END X509 CERTIFICATE----
```

Trusted certificates have the following lines:

```
-----BEGIN TRUSTED CERTIFICATE----
-----END TRUSTED CERTIFICATE----
```

The conversion to UTF8 format used with the name options assumes that T61Strings use the ISO8859-1 character set. This is wrong, but Netscape and MSIE do this, as do many certificates. So, although this is incorrect, it is more likely to display the majority of certificates correctly.

The `fingerprint` option takes the digest of the `DER` encoded certificate. This is commonly called a fingerprint. Because of the nature of message digests, the fingerprint of a certificate is unique to that certificate. Two certificates with the same fingerprint can be considered to be the same.

The Netscape fingerprint uses MD5 whereas MSIE uses SHA1.

The `email` option searches the subject name and the subject alternative name extension. Only unique email addresses will be printed out. It will not print the same address more than once.

RESTRICTIONS

Extensions in certificates are not transferred to certificate requests and vice versa.

It is possible to produce invalid certificates or requests by specifying the wrong private key or using inconsistent options in some cases. These should be checked.

There should be options to explicitly set such things as start and end dates rather than an offset from the current time.

The code to implement the verify behavior described in the Trust Settings is under development. It describes the intended behavior rather than the current behavior.

EXAMPLES

In these examples the \ character means the example should be on one line.

Display the contents of a certificate:

```
openssl x509 -in cert.pem -noout -text
```

Display the certificate serial number:

```
openssl x509 -in cert.pem -noout -serial
```

Display the certificate subject name:

```
openssl x509 -in cert.pem -noout -subject
```

Display the certificate subject name in RFC2253 form:

```
openssl x509 -in cert.pem -noout -subject -nameopt RFC2253
```

Display the certificate subject name in oneline form on a terminal supporting UTF8:

```
openssl x509 -in cert.pem -noout -subject -nameopt oneline -nameopt -escmsb
```

Display the certificate MD5 fingerprint:

```
openssl x509 -in cert.pem -noout -fingerprint
```

Display the certificate SHA1 fingerprint:

```
openssl x509 -sha1 -in cert.pem -noout -fingerprint
```

Convert a certificate from PEM to DER format:

```
openssl x509 -in cert.pem -inform PEM -out cert.der -outform DER
```

Convert a certificate to a certificate request:

```
openssl x509 -x509toreq -in cert.pem -out req.pem -signkey key.pem
```

Convert a certificate request into a self signed certificate using extensions for a CA:

```
openssl x509 -req -in careq.pem -extfile openssl.cnf -extensions v3_ca \
-signkey key.pem -out cacert.pem
```

Sign a certificate request using the CA certificate above and add user certificate extensions:

```
openssl x509 -req -in req.pem -extfile openssl.cnf -extensions v3_usr \
-CA cacert.pem -CAkey key.pem -CAcreateserial
```

Set a certificate to be trusted for SSL client use and change set its alias to "Steve's Class 1 CA"

```
openssl x509 -in cert.pem -addtrust sslclient \
-alias "Steve's Class 1 CA" -out trust.pem
```

SEE ALSO

Commands: *req*, *ca*, *genrsa*, *gendsa*, *verify*

A Data Structures and Header Files

This appendix lists the header files and the data structures included in HP SSL for OpenVMS.

Header Files

- SSL.H
- SSL2.H
- SSL23.H
- SSL3.H
- TLS.H

SSL_CTX Structure

The `SSL_CTX` structure is defined in `ssl.h`.

```
struct ssl_ctx_st

{
SSL_METHOD *method;
unsigned long options;
unsigned long mode;

STACK_OF(SSL_CIPHER) *cipher_list;
/* same as above but sorted for lookup */
STACK_OF(SSL_CIPHER) *cipher_list_by_id;

struct x509_store_st /* X509_STORE */ *cert_store;
struct lhash_st /* LHASH */ *sessions;/* a set of SSL_SESSIONs */
/* Most session-ids that will be cached, default is
 * SSL_SESSION_CACHE_MAX_SIZE_DEFAULT. 0 is unlimited. */
unsigned long session_cache_size;
struct ssl_session_st *session_cache_head;
struct ssl_session_st *session_cache_tail;

/* This can have one of 2 values, ored together,
 * SSL_SESS_CACHE_CLIENT,
 * SSL_SESS_CACHE_SERVER,
 * Default is SSL_SESSION_CACHE_SERVER, which means only
 * SSL_accept which cache SSL_SESSIONS. */

int session_cache_mode;
```

```
/* If timeout is not 0, it is the default timeout value set
 * when SSL_new() is called.  This has been put in to make
 * life easier to set things up */

long session_timeout;

/* If this callback is not null, it will be called each
 * time a session id is added to the cache.  If this function
 * returns 1, it means that the callback will do a
 * SSL_SESSION_free() when it has finished using it.  Otherwise,
 * on 0, it means the callback has finished with it.
 * If remove_session_cb is not null, it will be called when
 * a session-id is removed from the cache.  After the call,
 * OpenSSL will SSL_SESSION_free() it. */


int (*new_session_cb)(struct ssl_st *ssl,SSL_SESSION *sess);
void (*remove_session_cb)(struct ssl_ctx_st *ctx,SSL_SESSION *sess);
SSL_SESSION *(*get_session_cb)(struct ssl_st *ssl,
unsigned char *data,int len,int *copy);
struct
{
int sess_connect;/* SSL new conn - started */
int sess_connect_renegotiate;/* SSL reneg - requested */
int sess_connect_good;/* SSL new conne/reneg - finished */
int sess_accept;/* SSL new accept - started */
int sess_accept_renegotiate;/* SSL reneg - requested */
int sess_accept_good;/* SSL accept/reneg - finished */
int sess_miss;/* session lookup misses  */
int sess_timeout;/* reuse attempt on timeouted session */
int sess_cache_full;/* session removed due to full cache */
int sess_hit;/* session reuse actually done */
int sess_cb_hit;/* session-id that was not

 * in the cache was
 * passed back via the callback.  This
 * indicates that the application is
 * supplying session-id's from other
 * processes - spooky :-) */

} stats;

int references;

void (*info_callback)();

/* if defined, these override the X509_verify_cert() calls */

int (*app_verify_callback)();
char *app_verify_arg; /* never used; should be void * */

/* default values to use in SSL structures */

struct cert_st /* CERT */ *cert;
int read_ahead;
```

```
int verify_mode;
int verify_depth;
unsigned int sid_ctx_length;
unsigned char sid_ctx[SSL_MAX_SID_CTX_LENGTH];
int (*default_verify_callback)(int ok,X509_STORE_CTX *ctx);

int purpose;/* Purpose setting */
int trust;/* Trust setting */

/* Default password callback. */

pem_password_cb *default_passwd_callback;

/* Default password callback user data. */

void *default_passwd_callback_userdata;

/* get client cert callback */

int (*client_cert_cb)(/* SSL *ssl, X509 **x509, EVP_PKEY **pkey */);

/* what we put in client cert requests */

STACK_OF(X509_NAME) *client_CA;

int quiet_shutdown;

CRYPTO_EX_DATA ex_data;

const EVP_MD *rsa_md5;/* For SSLv2 - name is 'ssl2-md5' */
const EVP_MD *md5;/* For SSLv3/TLSv1 'ssl3-md5' */
const EVP_MD *sha1;   /* For SSLv3/TLSv1 'ssl3->sha1' */

STACK_OF(X509) *extra_certs;
STACK_OF(SSL_COMP) *comp_methods; /* stack of SSL_COMP, SSLv3/TLSv1 */

};
```

SSL Structure

The SSL structure is defined in ssl.h.

```
struct ssl_st
{
/* protocol version
 * (one of SSL2_VERSION, SSL3_VERSION, TLS1_VERSION)
 */

int version;
int type; /* SSL_ST_CONNECT or SSL_ST_ACCEPT */

SSL_METHOD *method; /* SSLv3 */
```

```
/* There are 2 BIO's even though they are normally both the
 * same.  This is so data can be read and written to different
 * handlers */

#ifndef NO_BIO

BIO *rbio; /* used by SSL_read */
BIO *wbio; /* used by SSL_write */
BIO *bbio; /* used during session-id reuse to concatenate
    * messages */

#else

char *rbio; /* used by SSL_read */
char *wbio; /* used by SSL_write */
char *bbio;
#endif

/* This holds a variable that indicates what we were doing
 * when a 0 or -1 is returned.  This is needed for
 * non-blocking IO so we know what request needs re-doing when
 * in SSL_accept or SSL_connect */

int rwstate;

/* true when we are actually in SSL_accept() or SSL_connect() */

int in_handshake;
int (*handshake_func)();

/* Imagine that here's a boolean member "init" that is
 * switched as soon as SSL_set_{accept/connect}_state
 * is called for the first time, so that "state" and
 * "handshake_func" are properly initialized.  But as
 * handshake_func is == 0 until then, we use this
 * test instead of an "init" member.
 */

int server;/* are we the server side? - mostly used by SSL_clear*/
int new_session;/* 1 if we are to use a new session */
int quiet_shutdown;/* don't send shutdown packets */
int shutdown;/* we have shut things down, 0x01 sent, 0x02

 * for received */

int state;/* where we are */
int rstate;/* where we are when reading */

BUF_MEM *init_buf;/* buffer used during init */
int init_num;/* amount read/written */
int init_off;/* amount read/written */

/* used internally to point at a raw packet */

unsigned char *packet;
unsigned int packet_length;
struct ssl2_state_st *s2; /* SSLv2 variables */
```

```
struct ssl3_state_st *s3; /* SSLv3 variables */
int read_ahead;/* Read as many input bytes as possible
                * (for non-blocking reads) */

int hit;/* reusing a previous session */
int purpose;/* Purpose setting */
int trust;/* Trust setting */

/* crypto */

STACK_OF(SSL_CIPHER) *cipher_list;
STACK_OF(SSL_CIPHER) *cipher_list_by_id;

/* These are the ones being used, the ones in SSL_SESSION are
 * the ones to be 'copied' into these ones */

EVP_CIPHER_CTX *enc_read_ctx;/* cryptographic state */
const EVP_MD *read_hash;/* used for mac generation */
#ifndef NO_COMP
COMP_CTX *expand;/* uncompress */
#else

char *expand;
#endif

EVP_CIPHER_CTX *enc_write_ctx;/* cryptographic state */
const EVP_MD *write_hash;/* used for mac generation */
#ifndef NO_COMP


COMP_CTX *compress;/* compression */
#else
char *compress;
#endif

/* session info */
/* client cert? */
/* This is used to hold the server certificate used */

struct cert_st /* CERT */ *cert;

/* the session_id_context is used to ensure sessions are only reused
 * in the appropriate context */

unsigned int sid_ctx_length;
unsigned char sid_ctx[SSL_MAX_SID_CTX_LENGTH];

/* This can also be in the session once a session is established */

SSL_SESSION *session;

/* Used in SSL2 and SSL3 */
int verify_mode; /* 0 don't care about verify failure.
 * 1 fail if verify fails */

int verify_depth;
int (*verify_callback)(int ok,X509_STORE_CTX *ctx); /* fail if callback returns 0 */
void (*info_callback)(); /* optional informational callback */
```

```
int error;/* error bytes to be written */
int error_code;/* actual code */

SSL_CTX *ctx;

/* set this flag to 1 and a sleep(1) is put into all SSL_read()
 * and SSL_write() calls, good for nbio debuging :-) */

int debug;

/* extra application data */

long verify_result;
CRYPTO_EX_DATA ex_data;

/* for server side, keep the list of CA_dn we can use */

STACK_OF(X509_NAME) *client_CA;
int references;
unsigned long options; /* protocol behaviour */
unsigned long mode; /* API behaviour */
int first_packet;
int client_version;/* what was passed, used for

 * SSLv3/TLS rollback check */

};
```

SSL_METHOD Structure

The SSL_METHOD structure is defined in ssl.h.

```
/* Used to hold functions for SSLv2 or SSLv3/TLSv1 functions */

typedef struct ssl_method_st

{
int version;
int (*ssl_new)(SSL *s);
void (*ssl_clear)(SSL *s);
void (*ssl_free)(SSL *s);
int (*ssl_accept)(SSL *s);
int (*ssl_connect)(SSL *s);
int (*ssl_read)(SSL *s,void *buf,int len);
int (*ssl_peek)(SSL *s,void *buf,int len);
int (*ssl_write)(SSL *s,const void *buf,int len);
int (*ssl_shutdown)(SSL *s);
int (*ssl_renegotiate)(SSL *s);
int (*ssl_renegotiate_check)(SSL *s);
long (*ssl_ctrl)(SSL *s,int cmd,long larg,char *parg);
long (*ssl_ctx_ctrl)(SSL_CTX *ctx,int cmd,long larg,char *parg);
```

```
SSL_CIPHER *(*get_cipher_by_char)(const unsigned char *ptr);
int (*put_cipher_by_char)(const SSL_CIPHER *cipher,unsigned char *ptr);
int (*ssl_pending)(SSL *s);
int (*num_ciphers)(void);

SSL_CIPHER *(*get_cipher)(unsigned ncipher);
struct ssl_method_st *(*get_ssl_method)(int version);
long (*get_timeout)(void);
struct ssl3_enc_method *ssl3_enc; /* Extra SSLv3/TLS stuff */
int (*ssl_version)();
long (*ssl_callback_ctrl)(SSL *s, int cb_id, void (*fp)());
long (*ssl_ctx_callback_ctrl)(SSL_CTX *s, int cb_id, void (*fp)());

} SSL_METHOD;
```

SSL_SESSION Structure

The `SSL_SESSION` structure is defined in `ssl.h`.

```
/* Lets make this into an ASN.1 type structure as follows
* SSL_SESSION_ID ::= SEQUENCE {
*version INTEGER,-- structure version number
*SSLversion INTEGER,-- SSL version number
*Cipher OCTET_STRING,-- the 3 byte cipher ID
*Session_ID OCTET_STRING,-- the Session ID
*Master_key OCTET_STRING,-- the master key
*Key_Arg [ 0 ] IMPLICITOCTET_STRING,-- the optional Key argument
*Time [ 1 ] EXPLICITINTEGER,-- optional Start Time
*Timeout [ 2 ] EXPLICITINTEGER,-- optional Timeout ins seconds
*Peer [ 3 ] EXPLICITX509,-- optional Peer Certificate
*Session_ID_context [ 4 ] EXPLICIT OCTET_STRING,   -- the Session ID context
*Verify_result [ 5 ] EXPLICIT INTEGER    -- X509_V_... code for `Peer'
*Compression [6] IMPLICIT ASN1_OBJECT-- compression OID XXXXX
*}
* Look in ssl/ssl_asn1.c for more details
* I'm using EXPLICIT tags so I can read the damn things using asn1parse :-).
*/

typedef struct ssl_session_st

{
int ssl_version;/* what ssl version session info is
 * being kept in here? */

/* only really used in SSLv2 */

unsigned int key_arg_length;
unsigned char key_arg[SSL_MAX_KEY_ARG_LENGTH];
int master_key_length;
unsigned char master_key[SSL_MAX_MASTER_KEY_LENGTH];

/* session_id - valid? */
```

```
unsigned int session_id_length;
unsigned char session_id[SSL_MAX_SSL_SESSION_ID_LENGTH];

/* this is used to determine whether the session is being reused in
 * the appropriate context. It is up to the application to set this,
 * via SSL_new */

unsigned int sid_ctx_length;
unsigned char sid_ctx[SSL_MAX_SID_CTX_LENGTH];
int not_resumable;

/* The cert is the certificate used to establish this connection */

struct sess_cert_st /* SESS_CERT */ *sess_cert;

/* This is the cert for the other end.
 * On clients, it will be the same as sess_cert->peer_key->x509
 * (the latter is not enough as sess_cert is not retained
 * in the external representation of sessions, see ssl_asn1.c). */

X509 *peer;

/* when app_verify_callback accepts a session where the peer's certificate
 * is not ok, we must remember the error for session reuse: */

long verify_result; /* only for servers */

int references;
long timeout;
long time;
int compress_meth;/* Need to lookup the method */

SSL_CIPHER *cipher;

unsigned long cipher_id;/* when ASN.1 loaded, this

 * needs to be used to load
 * the 'cipher' structure */

STACK_OF(SSL_CIPHER) *ciphers; /* shared ciphers? */
CRYPTO_EX_DATA ex_data; /* application specific data */

/* These are used to make removal of session-ids more
 * efficient and to implement a maximum cache size. */

struct ssl_session_st *prev,*next;

} SSL_SESSION;
```

SSL_CIPHER Structure

The `SSL_CIPHER` structure is defined in `ssl.h`.

```
/* used to hold info on the particular ciphers used */

typedef struct ssl_cipher_st

{
int valid;
const char *name;/* text name */
unsigned long id;/* id, 4 bytes, first is version */
unsigned long algorithms;/* what ciphers are used */
unsigned long algo_strength;/* strength and export flags */
unsigned long algorithm2;/* Extra flags */
int strength_bits;/* Number of bits really used */
int alg_bits;/* Number of bits for algorithm */
unsigned long mask;/* used for matching */
unsigned long mask_strength;/* also used for matching */

} SSL_CIPHER;
```

BIO Structure

The `BIO` structure is defined in `bio.h`.

```
struct bio_st
{
BIO_METHOD *method;
/* bio, mode, argp, argi, argl, ret */
long (*callback)(struct bio_st *,int,const char *,int, long,long);
char *cb_arg; /* first argument for the callback */

int init;
int shutdown;
int flags;/* extra storage */

int retry_reason;

int num;
void *ptr;
struct bio_st *next_bio;/* used by filter BIOs */
struct bio_st *prev_bio;/* used by filter BIOs */
int references;
unsigned long num_read;
unsigned long num_write;

CRYPTO_EX_DATA ex_data;
};
```

X509 Structure

The `X509` structure is defined in `x509.h`.

```
typedef struct x509_st

{
X509_CINF *cert_info;
X509_ALGOR *sig_alg;
ASN1_BIT_STRING *signature;
int valid;
int references;
char *name;
CRYPTO_EX_DATA ex_data;

/* These contain copies of various extension values */

long ex_pathlen;
unsigned long ex_flags;
unsigned long ex_kusage;
unsigned long ex_xkusage;
unsigned long ex_nscert;
ASN1_OCTET_STRING *skid;
struct AUTHORITY_KEYID_st *akid;

#ifndef NO_SHA
unsigned char sha1_hash[SHA_DIGEST_LENGTH];
#endif

X509_CERT_AUX *aux;

} X509;
```

B Open Source Notices

OpenSSL Open Source License

Copyright (c) 1998-2003 The OpenSSL Project. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software must display the following acknowledgment:
   "This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact openssl-core@openssl.org.
5. Products derived from this software may not be called "OpenSSL" nor may "OpenSSL" appear in their names without prior written permission of the OpenSSL Project.
6. Redistributions of any form whatsoever must retain the following acknowledgment:
   "This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/)"

THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE OpenSSL PROJECT OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

This product includes cryptographic software written by Eric Young (eay@cryptsoft.com). This product includes software written by Tim Hudson (tjh@cryptsoft.com).

Original SSLeay License

Copyright (c) 1995-1998 Eric Young (eay@cryptsoft.com) All rights reserved.

Redistribution and use in source and binary forms, with or without modification,
are permitted provided that the following conditions are met:

This library is free for commercial and non-commercial use as long as
the following conditions are aheared to. The following conditions
apply to all code found in this distribution, be it the RC4, RSA,
lhash, DES, etc., code; not just the SSL code. The SSL documentation
included with this distribution is covered by the same copyright terms
except that the holder is Tim Hudson (tjh@cryptsoft.com).

Copyright remains Eric Young's, and as such any Copyright notices in
the code are not to be removed. If this package is used in a product,
Eric Young should be given attribution as the author of the parts of the library used.
This can be in the form of a textual message at program startup or
in documentation (online or textual) provided with the package.

Redistribution and use in source and binary forms, with or without modification,
are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software
   must display the following acknowledgement:
   "This product includes cryptographic software written by
   Eric Young (eay@cryptsoft.com)"
   The word 'cryptographic' can be left out if the rouines from the library
   being used are not cryptographic related :-).
4. If you include any Windows specific code (or a derivative thereof) from
   the apps directory (application code) you must include an acknowledgement:
   "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"

THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ''AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.

The licence and distribution terms for any publically available version or
derivative of this code cannot be changed. i.e. this code cannot simply be
copied and put under another distribution licence
[including the GNU Public Licence.]

Index

A

Applications
  building using 32-bit APIs, 19
  building using 64-bit APIs, 19
  compiling and linking, 19
Asymmetric encryption, 29
Authentication
  client, 28
  server, 28

B

Backward compatibility, 21
BF_cbc_encrypt function, 151
BF_cfb64_encrypt function, 151
BF_decrypt function, 151
BF_ecb_encrypt function, 151
BF_encrypt function, 151
BF_ofb64_encrypt function, 151
BF_options function, 151
BF_set_key function, 151
bio function, 98
BIO_append_filename function, 140
BIO_callback_ctrl function, 99
BIO_ctrl function, 99
BIO_ctrl_get_read_request function, 102, 133
BIO_ctrl_get_write_guarantee function, 133
BIO_ctrl_pending function, 99, 103
BIO_ctrl_reset_read_request function, 133
BIO_ctrl_wpending function, 99
BIO_debug_callback function, 147
BIO_destroy_bio_pair function, 133
BIO_do_accept function, 130
BIO_do_connect function, 135
BIO_eof function, 99
BIO_f_base64 function, 104
BIO_f_buffer function, 106
BIO_f_cipher function, 108
BIO_f_md function, 110
BIO_f_null function, 113
BIO_f_ssl function, 114
BIO_find_type function, 120
BIO_flush function, 99
BIO_free all function, 122
BIO_free function, 122
BIO_get_accept_port function, 130
BIO_get_bind_mode function, 130
BIO_get_callback function, 147
BIO_get_callback_arg function, 147
BIO_get_cipher_ctx function, 108
BIO_get_cipher_status function, 108
BIO_get_close function, 99
BIO_get_conn_hostname function, 135
BIO_get_conn_int_port function, 135
BIO_get_conn_ip function, 135
BIO_get_conn_port function, 135
BIO_get_fd function, 138
BIO_get_fp function, 140
BIO_get_info_callback function, 99
BIO_get_md function, 110
BIO_get_md_ctx function, 110
BIO_get_mem_data function, 143
BIO_get_mem_ptr function, 143
BIO_get_num_renegotiates function, 114
BIO_get_read_request function, 133
BIO_get_retry_BIO function, 149
BIO_get_retry_reason function, 149
BIO_get_ssl function, 114
BIO_get_write_buf_size function, 133
BIO_get_write_guarantee function, 133
BIO_gets function, 128
BIO_int_ctrll function, 99
BIO_make_bio_pair function, 133
BIO_new function, 122
BIO_new_bio_pair function, 124, 133
BIO_new_buffer_ssl_connect function, 114
BIO_new_fd function, 138
BIO_new_file function, 140
BIO_new_fp function, 140
BIO_new_mem_buf function, 143
BIO_new_socket function, 146
BIO_new_ssl function, 114
BIO_new_ssl_connect function, 114
BIO_next function, 120
BIO_pending function, 99
BIO_pop function, 126
BIO_ptr_ctrl function, 99
BIO_push function, 126
BIO_puts function, 128
BIO_read function, 128
BIO_read_filename function, 140
BIO_reset function, 99
BIO_retry_type function, 149
BIO_rw_filename function, 140
BIO_s_accept function, 130
BIO_s_bio function, 133
BIO_s_connect function, 135
BIO_s_fd function, 138
BIO_s_file function, 140
BIO_s_mem function, 143
BIO_s_null function, 145
BIO_s_socket function, 146
BIO_seek function, 99
BIO_set function, 122
BIO_set_accept_bios function, 130
BIO_set_accept_port function, 130
BIO_set_bind_mode function, 130
BIO_set_callback function, 147
BIO_set_callback_arg function, 147
BIO_set_cipher function, 108
BIO_set_close function, 99
BIO_set_conn_hostname function, 135
BIO_set_conn_int_port function, 135
BIO_set_conn_ip function, 135
BIO_set_conn_port function, 135
BIO_set_fd function, 138
BIO_set_fp function, 140
BIO_set_info_callback function, 99
BIO_set_md function, 110
BIO_set_mem_buf function, 143
BIO_set_mem_eof_return function, 143
BIO_set_nbio function, 130, 135
BIO_set_nbio_accept function, 130
BIO_set_ssl function, 114
BIO_set_ssl_mode function, 114
BIO_set_ssl_renegotiate_bytes function, 114
BIO_set_ssl_renegotiate_timeout function, 114
BIO_set_write_buf_size function, 133

Index

BIO_should_io_special function, 149
BIO_should_read function, 149
BIO_should_retry function, 149
BIO_should_write function, 149
BIO_shutdown_wr function, 133
BIO_ssl_copy_session_id function, 114
BIO_ssl_shutdown function, 114
BIO_tell function, 99
BIO_vfree function, 122
BIO_wpending function, 99
BIO_write function, 128
BIO_write_filename function, 140
blowfish function, 151
bn function, 153
BN_add function, 160
BN_add_word function, 162
bn_add_words function, 173
BN_bin2bn function, 164
BN_bn2bin function, 164
BN_bn2dec function, 164
BN_bn2hex function, 164
BN_bn2mpi function, 164
bn_check_top function, 173
BN_clear function, 184
BN_clear_bit function, 187
BN_clear_free function, 184
BN_cmp function, 166
bn_cmp_words function, 173
BN_copy function, 168
BN_CTX_end function, 170
BN_CTX_free function, 169
BN_CTX_get function, 170
BN_CTX_init function, 169
BN_CTX_new function, 169
BN_CTX_start function, 170
BN_dec2bn function, 164
BN_div function, 160
BN_div_recp function, 182
BN_div_word function, 162
bn_div_words function, 173
bn_dump function, 173
BN_dup function, 168
BN_exp function, 160
bn_expand function, 173
bn_expand2 function, 173
bn_fix_top function, 173
BN_free function, 184
BN_from_montgomery function, 180
BN_gcd function, 160
BN_generate_prime function, 171
BN_get_word function, 189
BN_hex2bn function, 164
BN_init function, 184
BN_is_bit_set function, 187
BN_is_odd function, 166
BN_is_one function, 166
BN_is_prime function, 171
BN_is_prime_fasttest function, 171
BN_is_word function, 166
BN_is_zero function, 166
BN_lshift function, 187
BN_lshift1 function, 187
BN_mask_bits function, 187
BN_mod function, 160
BN_mod_exp function, 160
BN_mod_inverse function, 179
BN_mod_mul function, 160
BN_mod_mul_montgomery function, 180
BN_mod_mul_reciprocal function, 182
BN_mod_word function, 162
BN_MONT_CTX_copy function, 180
BN_MONT_CTX_free function, 180
BN_MONT_CTX_init function, 180
BN_MONT_CTX_new function, 180
BN_MONT_CTX_set function, 180
BN_mpi2bn function, 164
BN_mul function, 160
bn_mul_add_words function, 173
bn_mul_comba4 function, 173
bn_mul_comba8 function, 173
bn_mul_high function, 173
bn_mul_low_normal function, 173
bn_mul_low_recursive function, 173
bn_mul_normal function, 173
bn_mul_part_recursive function, 173
bn_mul_recursive function, 173
BN_mul_word function, 162
bn_mul_words function, 173
BN_new function, 184
BN_num_bits function, 185
BN_num_bits_word function, 185
BN_num_bytes function, 185
BN_one function, 189
BN_print function, 164
bn_print function, 173
BN_print_fp function, 164
BN_pseudo_rand function, 186
BN_rand function, 186
BN_RECP_CTX_free function, 182
BN_RECP_CTX_init function, 182
BN_RECP_CTX_new function, 182
BN_RECP_CTX_set function, 182
BN_rshift function, 187
BN_rshift1 function, 187
BN_set_bit function, 187
bn_set_high function, 173
bn_set_low function, 173
bn_set_max function, 173
BN_set_word function, 189
BN_sqr function, 160
bn_sqr_comba4 function, 173
bn_sqr_comba8 function, 173
bn_sqr_normal function, 173
bn_sqr_recursive function, 173
bn_sqr_words function, 173
BN_sub function, 160
BN_sub_word function, 162
bn_sub_words function, 173
BN_to_montgomery function, 180
BN_ucmp function, 166
BN_value_one function, 189
bn_wexpand function, 173
BN_zero function, 189
BUF_MEM_free function, 191
BUF_MEM_grow function, 191
BUF_MEM_new function, 191
BUF_strdup function, 191

C

ca, 193
ca.pl, 200
CDSA
  definition of, 27
Certificate, 29
  client request, 36
  command procedure to set up example programs, 85
  configuring in the client and server, 48
  formats, 52
  installing, 38
  intermediate, 41
  loading, 55
  peer, 61
  request file, 35
  revoking, 43
  self-signed, 38
  server request, 36
  signing request, 36
  X509, 41
Certificate authorities, 29
Certificate chain, 41
Certificate Revocation List, 43
Certificate tool, 33
Cipher commands, 68
Ciphers, 30
Command line interface (CLI), 65
CRL, 43
crypto function, 213
CRYPTO_destroy_dynlockid function, 579
CRYPTO_get_ex_data function, 214
CRYPTO_get_new_dynlockid function, 579
CRYPTO_lock function, 579
CRYPTO_num_locks function, 579
CRYPTO_set_dynlock_create_callback function, 579
CRYPTO_set_dynlock_destroy_callback function, 579
CRYPTO_set_dynlock_lock_callback function, 579
CRYPTO_set_ex_data function, 214
CRYPTO_set_id_callback function, 579
CRYPTO_set_locking_callback function, 579

D

d2i_DHparams function, 215
d2i_Netscape_RSA function, 216
d2i_RSAPrivateKey function, 216
d2i_RSAPublicKey function, 216
d2i_SSL_SESSION function, 217
Data structures, 45
  APIs used for creating and deallocating, 45
Data transmission, 61
DER certificate format, 52
des_cbc_cksum function, 218
des_cfb_encrypt function, 218
des_cfb64_encrypt function, 218
des_crypt function, 218
des_ecb_encrypt function, 218
des_ecb2_encrypt function, 218
des_ecb3_encrypt function, 218
des_ede2_cbc_encrypt function, 218
des_ede2_cfb64_encrypt function, 218
des_ede2_ofb64_encrypt function, 218
des_ede3_cbc_encrypt function, 218
des_ede3_cbcm_encrypt function, 218
des_ede3_cfb64_encrypt function, 218
des_ede3_ofb64_encrypt function, 218
des_enc_read function, 218
des_enc_write function, 218
des_fcrypt function, 218
des_is_weak_key function, 218
des_key_sched function, 218
des_ncbc_encrypt function, 218
des_ofb_encrypt function, 218
des_ofb64_encrypt function, 218
des_pcbc_encrypt function, 218
des_quad_cksum function, 218
des_random_key function, 218
des_read_2passwords function, 218
des_read_password function, 218
des_read_pw_string function, 218
des_set_key function, 218
des_set_key_checked function, 218
des_set_key_unchecked function, 218
des_set_odd_parity function, 218
des_string_to_2keys function, 218
des_xcbc_encrypt function, 218
dh function, 230
DH parameter file, 71
DH_check function, 233
DH_compute_key function, 232
DH_free function, 236
DH_generate_key function, 232
DH_generate_parameters function, 233
DH_get_default_openssl_method function, 237
DH_get_ex_data function, 235
DH_get_ex_new_index function, 235
DH_new function, 236
DH_new_method function, 237
DH_OpenSSL function, 237
DH_set_default_openssl_method function, 237
DH_set_ex_data function, 235
DH_set_method function, 237
DH_size function, 239
DHparams_print function, 380
DHparams_print_fp function, 380
Digital signature, 30, 31
Directory format for UNIX and OpenVMS, 16
Directory structure for SSL, 19
Disk space requirements, 15
DSA certificate, 71
dsa function, 240
DSA key, 71
DSA_do_sign function, 244
DSA_do_verify function, 244
DSA_dup_DH function, 245
DSA_free function, 250
DSA_generate_key function, 246
DSA_generate_parameters function, 247
DSA_get_default_openssl_method function, 251
DSA_get_ex_data function, 249
DSA_get_ex_new_index function, 249
DSA_new function, 250
DSA_new_method function, 251
DSA_OpenSSL function, 251
DSA_print function, 380
DSA_print_fp function, 380

DSA_set_default_openssl_method function, 251
DSA_set_ex_data function, 249
DSA_set_method function, 251
DSA_SIG_free function, 254
DSA_SIG_new function, 254
DSA_sign function, 255
DSA_sign_setup function, 255
DSA_size function, 256
DSA_verify function, 255
DSAparams_print function, 380
DSAparams_print_fp function, 380

E

Encoding commands, 68
Encryption, 29
err function, 263
ERR_add_error_data function, 277
ERR_clear_error function, 267
ERR_error_string function, 268
ERR_error_string_n function, 268
ERR_free_strings function, 273
ERR_func_error_string function, 268
ERR_get_error function, 270
ERR_get_error_line function, 270
ERR_get_error_line_data function, 270
ERR_GET_FUNC function, 272
ERR_GET_LIB function, 272
ERR_get_next_error_library function, 275
ERR_GET_REASON function, 272
ERR_lib_error_string function, 268
ERR_load_crypto_strings function, 273, 274
ERR_load_SSL_strings function, 274
ERR_load_strings function, 275
ERR_PACK function, 275
ERR_peek_error function, 270
ERR_peek_error_line function, 270
ERR_peek_error_line_data function, 270
ERR_print_errors function, 276
ERR_print_errors_fp function, 276
ERR_put_error function, 277
ERR_reason_error_string function, 268
ERR_remove_state function, 278
evp function, 279
EVP_CIPHER_asn1_to_param function, 284
EVP_CIPHER_block_size function, 284
EVP_CIPHER_CTX_block_size function, 284
EVP_CIPHER_CTX_cipher function, 284
EVP_CIPHER_CTX_cleanup function, 284
EVP_CIPHER_CTX_ctrl function, 284
EVP_CIPHER_CTX_flagst function, 284
EVP_CIPHER_CTX_get_app_data function, 284
EVP_CIPHER_CTX_iv_length function, 284
EVP_CIPHER_CTX_key_length function, 284
EVP_CIPHER_CTX_mode function, 284
EVP_CIPHER_CTX_nid function, 284
EVP_CIPHER_CTX_set_app_data function, 284
EVP_CIPHER_CTX_set_key_length function, 284
EVP_CIPHER_CTX_type function, 284
EVP_CIPHER_flags function, 284
EVP_CIPHER_iv_length function, 284
EVP_CIPHER_key_length function, 284
EVP_CIPHER_mode function, 284
EVP_CIPHER_nid function, 284
EVP_CIPHER_param_to_asn1 function, 284
EVP_CIPHER_type function, 284
EVP_CipherFinal function, 284
EVP_CipherInit function, 284
EVP_CipherUpdate function, 284
EVP_DecryptFinal function, 284
EVP_DecryptInit function, 284
EVP_DecryptUpdate function, 284
EVP_DigestInit function, 280
EVP_DigestUpdate function, 280
EVP_dss function, 280
EVP_dss1 function, 280
EVP_EncryptFinal function, 284
EVP_EncryptInit function, 284
EVP_EncryptUpdate function, 284
EVP_get_cipherbyname function, 284
EVP_get_cipherbynid function, 284
EVP_get_cipherbyobj function, 284
EVP_get_digestbyname function, 280
EVP_get_digestbynid function, 280
EVP_get_digestbyobj function, 280
EVP_MAX_MD_SIZE function, 280
EVP_MD_block_size function, 280
EVP_MD_CTX_block_size function, 280
EVP_MD_CTX_copy function, 280
EVP_MD_CTX_md function, 280
EVP_MD_CTX_size function, 280
EVP_MD_CTX_type function, 280
EVP_md_null function, 280
EVP_MD_pkey_type function, 280
EVP_MD_size function, 280
EVP_MD_type function, 280
EVP_md2 function, 280
EVP_md5 function, 280
EVP_mdc2 function, 280
EVP_OpenFinal function, 291
EVP_OpenInit function, 291
EVP_OpenUpdate function, 291
EVP_ripemd160 function, 280
EVP_SealFinal function, 292
EVP_SealInit function, 292
EVP_SealUpdate function, 292
EVP_sha function, 280
EVP_sha1 function, 280
EVP_SignFinal function, 294
EVP_SignInit function, 294
EVP_SignUpdate function, 294
EVP_VerifyFinal function, 296
EVP_VerifyInit function, 296
EVP_VerifyUpdate function, 296

H

Handshake, 28
  performing on server and client, 60
  renegotiating, 63
Hardware requirements, 15
Hash function, 30
HMAC function, 301
HMAC_cleanup function, 301
HMAC_Final function, 301
HMAC_Init function, 301
HMAC_Update function, 301

I

i2d_DHparams function, 215
i2d_Netscape_RSA function, 216
i2d_RSAPrivateKey function, 216
i2d_RSAPublicKey function, 216
i2d_SSL_SESSION function, 217
Installing
  PCSI command, 16
  postinstallation tasks, 18
  stopping and restarting, 18

K

Key file, 71

L

lh_delete function, 305
lh_doall function, 305
lh_doall_arg function, 305
lh_error function, 305
lh_free function, 305
lh_insert function, 305
lh_new function, 305
lh_node_stats function, 303
lh_node_stats_bio function, 303
lh_node_usage_stats function, 303
lh_node_usage_stats_bio function, 303
lh_retrieve function, 305
lh_stats function, 303
lh_stats_bio function, 303
lhash function, 305
Logical names
  command to set up, 18

M

MD2 function, 308
MD2_Final function, 308
MD2_Init function, 308
MD2_Update function, 308
MD4 function, 308
MD4_Final function, 308
MD4_Init function, 308
MD4_Update function, 308
MD5 function, 308
MD5_Final function, 308
MD5_Init function, 308
MD5_Update function, 308
MDC2 function, 310
MDC2_Final function, 310
MDC2_Init function, 310
MDC2_Update function, 310
Message digest commands, 68
modes device special file, 225
MultiNet, 15

N

NET certificate format, 52

O

One-way hash function, 30
Open Group, 16
OpenSSL command line interface (CLI), 65
OpenSSL commands
  encoding and cipher, 68
  message digest, 68
  pseudo, 65
  standard, 66
OpenSSL_add_all_algorithms function, 318
OpenSSL_add_all_ciphers function, 318
OpenSSL_add_all_digests function, 318
OpenSSL_add_ssl_algorithms function, 535
OPENSSL_VERSION_NUMBER function, 320
Options file, 19

P

Passphrase arguments, 71
PEM certificate format, 52
pem function, 324
pkcs12 utility, 336
Prerequisites
  disk space, 15
  hardware, 15
  software, 15
Private key encryption, 29
Pseudo commands, 65
Public key encryption, 29

R

RAND_add function, 347
RAND_bytes function, 349
RAND_cleanup function, 350
RAND_egd function, 351
RAND_event function, 347
RAND_file_name function, 352
RAND_get_rand_method function, 353
RAND_load_file function, 352
RAND_pseudo_bytes function, 349
RAND_screen function, 347
RAND_seed function, 347
RAND_set_rand_method function, 353
rand_ssl function, 355
RAND_SSLeay function, 353
RAND_status function, 347
RAND_write_file function, 352
RC4 function, 358
RC4_set_key function, 358
Release notes, 20
RIPEMD160 function, 367
RIPEMD160_Final function, 367
RIPEMD160_Init function, 367
RIPEMD160_Update function, 367
Root CA, 48
rsa function, 369
RSA_blinding_off function, 372
RSA_blinding_on function, 372
RSA_check_key function, 373
RSA_flags function, 385
RSA_free function, 377
RSA_generate_key function, 374
RSA_get_default_openssl_method function, 385
RSA_get_ex_data function, 375
RSA_get_ex_new_index function, 375
RSA_get_method function, 385
RSA_new function, 377
RSA_new_method function, 385

RSA_null_method function, 385
RSA_padding_add_none function, 378
RSA_padding_add_PKCS1_OAEP function, 378
RSA_padding_add_PKCS1_type_1 function, 378
RSA_padding_add_PKCS1_type_2 function, 378
RSA_padding_add_SSLv23 function, 378
RSA_padding_check_none function, 378
RSA_padding_check_PKCS1_OAEP function, 378
RSA_padding_check_PKCS1_type_1 function, 378
RSA_padding_check_PKCS1_type_2 function, 378
RSA_padding_check_SSLv23 function, 378
RSA_PKCS1_RSAref function, 385
RSA_PKCS1_SSLeay function, 385
RSA_print function, 380
RSA_print_fp function, 380
RSA_private_decrypt function, 383
RSA_private_encrypt function, 382
RSA_public_decrypt function, 382
RSA_public_encrypt function, 383
RSA_set_default_openssl_method function, 385
RSA_set_ex_data function, 375
RSA_set_method function, 385
RSA_sign function, 389
RSA_sign_ASN1_OCTET_STRING function, 390
RSA_size function, 391
RSA_verify function, 389
RSA_verify_ASN1_OCTET_STRING function, 390

S

s_client command, 395
s_server utility, 398
SHA1 function, 405
SHA1_Final function, 405
SHA1_Init function, 405
SHA1_Update function, 405
Shareable image filenames, 19
Software requirements, 15
SSL
  definition of, 27
SSL client authentication, 28
SSL handshake, 28
SSL library, 416
SSL Protocol, 27
SSL server authentication, 28
SSL shareable image filenames, 19
SSL$EXAMPLES_SETUP.TEMPLATE, 85
SSL$UTILS.COM, 65
SSL_accept function, 426
SSL_add_client_CA function, 471
SSL_add_session function, 448
SSL_alert_desc_string function, 428, 430
SSL_alert_desc_string_long function, 428, 430
SSL_alert_type_string function, 430
SSL_alert_type_string_long function, 430
SSL_callback_ctrl function, 435, 450
SSL_check_private_key function, 437, 502
SSL_CIPHER_description function, 438
SSL_CIPHER_get_bits function, 438
SSL_CIPHER_get_name function, 438
SSL_CIPHER_get_version function, 438
SSL_clear function, 440
SSL_COMP_add_compression_method function, 441
SSL_connect function, 442
SSL_copy_session_id function, 444
SSL_ctrl function, 445, 450
SSL_CTX_add_client_CA function, 471
SSL_CTX_add_extra_chain_cert function, 447
SSL_CTX_add_session function, 448
SSL_CTX_callback_ctrl function, 450
SSL_CTX_check_private_key function, 502
SSL_CTX_ctrl function, 450
SSL_CTX_flush_sessions function, 451
SSL_CTX_free function, 452
SSL_CTX_get_cert_store function, 453
SSL_CTX_get_client_CA_list function, 512
SSL_CTX_get_ex_data function, 454
SSL_CTX_get_ex_new_index function, 454
SSL_CTX_get_info_callback function, 476
SSL_CTX_get_mode function, 479
SSL_CTX_get_options function, 481
SSL_CTX_get_quiet_shutdown function, 455
SSL_CTX_get_session_cache_mode function, 486
SSL_CTX_get_timeout function, 491
SSL_CTX_get_verify_callback function, 456
SSL_CTX_get_verify_depth function, 456
SSL_CTX_get_verify_mode function, 456
SSL_CTX_load_verify_locations function, 458
SSL_CTX_new function, 460
SSL_CTX_remove_session function, 448
SSL_CTX_sess_accept function, 462
SSL_CTX_sess_accept_good function, 462
SSL_CTX_sess_accept_renegotiate function, 462
SSL_CTX_sess_cache_full function, 462
SSL_CTX_sess_cb_hits function, 462
SSL_CTX_sess_connect function, 462
SSL_CTX_sess_connect_good function, 462
SSL_CTX_sess_connect_renegotiate function, 462
SSL_CTX_sess_get_cache_size function, 464
SSL_CTX_sess_get_get_cb function, 465
SSL_CTX_sess_get_new_cb function, 465
SSL_CTX_sess_get_remove_cb function, 465
SSL_CTX_sess_hits function, 462
SSL_CTX_sess_misses function, 462
SSL_CTX_sess_number function, 462
SSL_CTX_sess_set_cache_size function, 464
SSL_CTX_sess_set_get_cb function, 465
SSL_CTX_sess_set_new_cb function, 465
SSL_CTX_sess_set_remove_cb function, 465
SSL_CTX_sess_timeouts function, 462
SSL_CTX_sessions function, 467
SSL_CTX_set_cert_store function, 468
SSL_CTX_set_cert_verify_cb function, 469
SSL_CTX_set_cipher_list function, 470
SSL_CTX_set_client_CA_list function, 471
SSL_CTX_set_def_verify_paths function, 473
SSL_CTX_set_default_passwd_cb function, 474
SSL_CTX_set_default_passwd_cb_userdata
    function, 474
SSL_CTX_set_ex_data function, 454
SSL_CTX_set_info_callback function, 476
SSL_CTX_set_mode function, 479
SSL_CTX_set_options function, 481
SSL_CTX_set_purpose function, 484
SSL_CTX_set_quiet_shutdown function, 485
SSL_CTX_set_session_cache_mode function, 486
SSL_CTX_set_session_id_context function, 488
SSL_CTX_set_ssl_version function, 490
SSL_CTX_set_timeout function, 491

SSL_CTX_set_tmp_dh function, 492
SSL_CTX_set_tmp_dh_callback function, 492
SSL_CTX_set_tmp_rsa_callback function, 495
SSL_CTX_set_verify function, 497
SSL_CTX_set_verify_depth function, 497
SSL_CTX_use_certificate function, 502
SSL_CTX_use_certificate_ASN1 function, 502
SSL_CTX_use_certificate_chain_file function, 502
SSL_CTX_use_certificate_file function, 502
SSL_CTX_use_PrivateKey function, 502
SSL_CTX_use_PrivateKey_ASN1 function, 502
SSL_CTX_use_PrivateKey_file function, 502
SSL_CTX_use_RSAPrivateKey function, 502
SSL_CTX_use_RSAPrivateKey_ASN1 function, 502
SSL_CTX_use_RSAPrivateKey_file function, 502
SSL_flush_sessions function, 451
SSL_free function, 509
SSL_get_accept_state function, 555
SSL_get_cipher function, 513
SSL_get_cipher_bits function, 513
SSL_get_cipher_list function, 511
SSL_get_cipher_name function, 513
SSL_get_cipher_version function, 513
SSL_get_ciphers function, 511
SSL_get_client_CA_list function, 512
SSL_get_current_cipher function, 513
SSL_get_default_timeout function, 514
SSL_get_error function, 515
SSL_get_ex_data function, 518
SSL_get_ex_data_X509_STORE_CTX_idx function, 517
SSL_get_ex_new_index function, 518
SSL_get_fd function, 519
SSL_get_info_callback function, 476
SSL_get_mode function, 479
SSL_get_options function, 481
SSL_get_peer_cert_chain function, 522
SSL_get_peer_certificate function, 523
SSL_get_rbio function, 527
SSL_get_session function, 529
SSL_get_shared_ciphers, 531
SSL_get_shutdown function, 562
SSL_get_SSL_CTX function, 532
SSL_get_ssl_method function, 490
SSL_get_verify_callback function, 456
SSL_get_verify_depth function, 456
SSL_get_verify_mode function, 456
SSL_get_verify_result function, 533
SSL_get_version function, 534
SSL_library_init function, 535
SSL_load_client_CA_file function, 536
SSL_load_error_strings function, 273, 274
SSL_new function, 537
SSL_pending function, 539
SSL_read function, 540
SSL_remove_session function, 448
SSL_rstate_string function, 543
SSL_rstate_string_long function, 543
SSL_SESSION_free function, 545
SSL_SESSION_get_ex_data function, 546
SSL_SESSION_get_ex_new_index function, 546
SSL_SESSION_get_time function, 548
SSL_SESSION_get_timeout function, 548
SSL_SESSION_print function, 552
SSL_SESSION_print_fp function, 552
SSL_session_reused function, 553
SSL_SESSION_set_ex_data function, 546
SSL_SESSION_set_time function, 548
SSL_SESSION_set_timeout function, 548
SSL_set_bio function, 554
SSL_set_cipher_list function, 470
SSL_set_client_CA_list function, 471
SSL_set_connect_state function, 555
SSL_set_ex_data function, 518
SSL_set_fd function, 556
SSL_set_info_callback function, 476
SSL_set_mode function, 479
SSL_set_options function, 481
SSL_set_session function, 561
SSL_set_session_id_context function, 488
SSL_set_shutdown function, 562
SSL_set_ssl_method function, 490
SSL_set_tmp_dh function, 492
SSL_set_tmp_dh_callback function, 492
SSL_set_tmp_rsa_callback function, 495
SSL_set_verify function, 497
SSL_set_verify_depth function, 497
SSL_set_verify_result function, 564
SSL_shutdown function, 565
SSL_state, 566
SSL_state_string, 566, 573
SSL_state_string_long function, 573
SSL_use_certificate function, 502
SSL_use_certificate_ASN1 function, 502
SSL_use_certificate_file function, 502
SSL_use_PrivateKey function, 502
SSL_use_PrivateKey_ASN1 function, 502
SSL_use_PrivateKey_file function, 502
SSL_use_RSAPrivateKey function, 502
SSL_use_RSAPrivateKey_ASN1 function, 502
SSL_use_RSAPrivateKey_file function, 502
SSL_want function, 575
SSL_want_nothing function, 575
SSL_want_read function, 575
SSL_want_write function, 575
SSL_want_x509_lookup function, 575
SSL_write function, 577
SSLeay function, 320
SSLeay_add_ssl_algorithms function, 535
SSLeay_version function, 320
Standard commands, 66
Symbols
  command to set up, 18

T

TCP/IP connection
  setting up, 58
TCP/IP Services for OpenVMS, 15
TCPware, 15

U

UNIX directory format, 16