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

HP SSL Version 1.3 for OpenVMS

OpenVMS I64 Version 8.2 or higher
OpenVMS Alpha Version 7.3-2 or higher

This manual supersedes *HP Open Source Security for OpenVMS HP SSL for OpenVMS*, Version 8.2

Manufacturing Part Number: BA554-90007
July 2006

Legal Notice

ZK6661

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

1. Installation and Release Notes

Tables

Figures

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 Industry Standard 64 and OpenVMS Alpha.

For information about HP SSL for OpenVMS VAX, see the *HP Open Source Security for OpenVMS, Volume 2: HP SSL for OpenVMS* for HP SSL Version 1.2.

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 OpenVMS I64, 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 lists the example programs included in the HP SSL kit.

Chapter 6 describes the OpenSSL command line interface.

The OpenSSL Command Line Interface (CLI) Reference describes the command line interface that allows you to use the cryptography functions of SSL's cryptography library from the OpenSSL command prompt.

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

The 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.

Convention	Meaning
[ ]	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.
|	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.
{ }	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.
bold type	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.
italic type	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).
UPPERCASE TYPE	Uppercase type indicates a command, the name of a routine, the name of a file, or the abbreviation for a system privilege.
`Example`	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.
–	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.
numbers	All numbers in text are assumed to be decimal unless otherwise noted. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated.

1 Installation and Release Notes

This chapter contains hardware and software prerequisites, installation instructions, postinstallation tasks, instructions for building your application, the HP SSL directory structure, and release notes for HP SSL Version 1.3 for OpenVMS. For an overview of HP SSL, see Chapter 2.

The information in this chapter applies to HP SSL running on OpenVMS I64 and OpenVMS Alpha. For information about HP SSL for OpenVMS VAX, see the *HP Open Source Security for OpenVMS, Volume 2: HP SSL for OpenVMS* for HP SSL Version 1.2.

1.1 Installation Requirements and Prerequisites

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

1.1.1 Hardware Prerequisites

Disk Space Requirements

The HP 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.

1.1.2 Software Prerequisites

HP SSL for OpenVMS requires the following software.

Operating System

HP OpenVMS Alpha Version 7.3-2 or higher, or

HP OpenVMS Industry Standard 64 Version 8.2 or higher

TCP/IP Transport

HP TCP/IP Services for OpenVMS Version 5.6 or higher (for HP SSL on OpenVMS I64 and OpenVMS Alpha Version 8.2 or higher), or

HP TCP/IP Services for OpenVMS Version 5.5 or higher (for HP SSL on OpenVMS Alpha Version 7.3-2)

NOTE HP SSL for OpenVMS has been tested and verified using HP TCP/IP Services for OpenVMS. On OpenVMS Alpha, there are no known problems running HP SSL for OpenVMS with other TCP/IP network products, including TCPware and MultiNet from Process Software Corporation. However, HP has not formally tested and verified these other products.

1.1.3 Account Quotas and System Parameters

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

1.1.4 New Features in HP SSL Version 1.3 for OpenVMS

HP SSL Version 1.3 for OpenVMS, based on OpenSSL 0.9.7e, is included in OpenVMS Version 8.3. (The previous version of HP SSL was based on OpenSSL 0.9.7d.)

New features in HP SSL Version 1.3 include:

- HP SSL Version 1.3 is now included in the OpenVMS operating system as a SIP (system integrated product) . SSL for OpenVMS is installed automatically when you install or upgrade to OpenVMS Version 8.3.
- Bug Fixes in OpenSSL 0.9.7e
  - — Fixed race condition when CRLs are checked in a multithreaded environment.
  - — Added Delta CRL to extension code.
  - — Fixed s3_pkt.c so alerts are sent properly.
  - — Reduced chances of duplicate issuer name and serial numbers (in violation of RFC3280) using the OpenSSL certificate creation utilities.
  - — Removed potential SSL Protocol 2.0 rollback.

    The functionality of SSL_OP_MSIE_SSLV2_RSA_PADDING (part of SSL_OP_ALL) has been removed from 0.9.7e. This option can be used to disable the countermeasure against man-in-the-middle protocol-version rollback in the SSL Protocol 2.0 server implementation. See http://www.openssl.org/news/secadv_20051011.txt for more information.

1.2 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.

1.3 Installing HP SSL for OpenVMS Automatically During OpenVMS Installation or Upgrade

HP SSL Version 1.3 is included in the OpenVMS operating system as a SIP (system integrated product). Previous versions of HP SSL were included in previous versions of OpenVMS as a layered product.

NOTE **SSL for OpenVMS is now installed automatically** when you install or upgrade to OpenVMS Version 8.3, and previous installed versions of HP SSL are automatically removed. You no longer need to install the PCSI file separately.

When the OpenVMS installation or upgrade procedure is complete, you must define the HP SSL foreign commands and (optionally) run the Certificate Tool before you use HP SSL. See Section 1.5 for more information.

1.4 Downloading and Installing HP SSL for OpenVMS from Web Site

You can install HP SSL Version 1.3 on versions of OpenVMS earlier than 8.3. A PCSI kit of HP SSL for OpenVMS is available for download from the HP SSL web site at

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

1.4.1 Before Installing HP SSL for OpenVMS

Beginning in HP SSL Version 1.3, the installation procedure **automatically removes** the previous version of HP SSL before installing the new version. For example, if you have Version 1.2 installed, it is removed during the installation procedure and the product removal is displayed in the installation log.

The HP SSL Version 1.3 installation procedure also **automatically removes** any old SSL kits that have a kit name beginning with DEC or CPQ. This removal is done silently during the preconfigure phase and is not shown in the installation log. For example, if you have SSL Version 1.1-B (kit name CPQ) installed, it is silently removed when you install SSL Version 1.3.

NOTE Do not use the PRODUCT REMOVE command to manually remove HP SSL Version 1.2 or higher. If you attempt to use PRODUCT REMOVE on these versions of HP SSL, you will receive a PCSI error that recommends terminating the operation. If you ignore the warning and continue to remove HP SSL, HP strongly recommends that you use PRODUCT INSTALL to install the HP SSL Version 1.3 PCSI kit as soon as possible. Other components in OpenVMS require that HP SSL is installed.

Before you begin the installation of HP SSL, perform the following steps:

1. *Preserve the SSL configuration files* OPENSSL-VMS.CNF and OPENSSL.CNF (if you modified them) by copying them to another disk and directory before installing HP SSL.
2. **Shut down HP SSL on each node in the cluster** before installing HP SSL on a common system disk in a cluster.

1.4.2 Installation Procedure

Install the HP SSL for OpenVMS kit by entering the following command:

```
$ PRODUCT INSTALL SSL
```

NOTE Beginning in HP SSL Version 1.3 for OpenVMS, HP SSL is always installed into SYS$SYSDEVICE:[VMS$COMMON]. The /DESTINATION qualifier is no longer supported.

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 deinstallation and installation procedures progress, the system displays information similar to the following output.

NOTE Specifying the `/HELP` qualifier on the `PRODUCT INSTALL` command line displays additional information about HP SSL.

```
$ PRODUCT INSTALL SSL/SOURCE=DKA500:[KITS] /HELP

The following product has been selected:

    HP AXPVMS SSL V1.3-281                 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.

HP AXPVMS SSL V1.3-281: SSL for OpenVMS Alpha V1.3 (Based on OpenSSL 0.9.7e)

    SSL for OpenVMS provides a toolkit that implements SSL V2/V3, TLS V1,
    and a general purpose cryptography library.

    © Copyright 2006 Hewlett-Packard Development Company, L.P.

    This software is installable on OpenVMS processors using the POLYCENTER
    Software Installation utility.

    IMPORTANT LEGAL NOTICE:

            Exports of this product are subject to U.S. Export Administration
            Regulations pertaining to encryption items and may require that
            individual export authorization be obtained from the U.S.
            Department of Commerce.


The /DESTINATION qualifier is not supported with SSL V1.3

As of SSL V1.3, the SSL product must be installed on the system disk.
If you specified a location other than the system disk with the use of the
qualifier /DESTINATION, it is recommended that you stop the installation
```

```
and restart it with the following command:

        $ PRODUCT INSTALL SSL

If you did not specify the /DESTINATION qualifier, answer NO to the
termination question, and continue with the installation.
Terminating is strongly recommended.  Do you want to terminate? [YES] NO

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:
    HP AXPVMS SSL V1.3-281                 DISK$DWLLNG_A_V73:[VMS$COMMON.]
The following product will be removed from destination:
    HP AXPVMS SSL V1.2                     DISK$DWLLNG_A_V73:[VMS$COMMON.]

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

The following product has been installed:
    HP AXPVMS SSL V1.3-281                 Layered Product
The following product has been removed:
    HP AXPVMS SSL V1.2                     Layered Product

%PCSI-I-IVPEXECUTE, executing test procedure for HP AXPVMS SSL V1.3-281 ...
%PCSI-I-IVPSUCCESS, test procedure completed successfully

HP AXPVMS SSL V1.3-281: SSL for OpenVMS Alpha V1.3 (Based on OpenSSL 0.9.7e)

    There are post installation tasks that you must complete
    including the following items that are described in detail:

        - ensuring SSL startup and logical names creation files
          are executed

        - updating or copying the necessary startup, shutdown and
          configuration files from the installed template files

        - running the Installation Verification Program (IVP)

    Refer to the SSL release notes and the OpenVMS SSL documentation for
    more information about activities that should be performed once the
    installation has finished.

    SSL has created the following directory structure and files 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.org provided documentation & information
```

```
    [SSL.INCLUDE]          Contains the C Header (.H) files
    [SSL.TEST]             Contains the files used during the IVP

    [SYS$STARTUP]          Startup and shutdown templates and files
    [SYSHLP]               Release notes
    [SYSHLP.EXAMPLES.SSL]  SSL crypto and secure session examples
    [SYSLIB]               SSL shareable image files
    [SYSTEST]              SSL$IVP.COM test files


...after upgrading from previous SSL versions...

The SSL release notes provide information to verify the SSL startup,
shutdown, and configuration template files.  Template files provide the
user with new features or changes, but do not overwrite existing command
procedures and configuration files.  A product upgrade or re-installation
will not overwrite or create a new file version if the file has been odified.
It will only create the template files. It is suggested that you review
these files for any changes.

For more information, refer to the SSL Release Notes and other SSL
files using the system logical name definitions, or the subdirectory of
the PCSI destination device and directory.

...including verifying startup command procedures and logical names...

Once the installation is complete, verify that SSL$STARTUP.COM is
located in SYS$MANAGER:SYSTARTUP_VMS.COM file. This will define the
SSL$ executive mode logical names in the SYSTEM logical name table,
and install the SSL shareable images in memory that reside in the
[SYSLIB] directory.

Also, add SSL$SHUTDOWN.COM to the SYS$MANAGER:SYSHUTDWN.COM file to remove
the installed images and deassign the SSL$ logical name definitions.

If you have customized the SSL command files for the site, it is
suggested that you compare the SSL provided template files with your
existing command procedures and take the appropriate action to update
your files. A product upgrade or re-installation will not overwrite
these files.

By default SYS$STARTUP: logical can be used to locate the SSL provided
startup files.

System managers should modify site-specific requirements in SSL files:

   SSL$COM:SSL$SYSTARTUP.COM
   SSL$COM:SSL$SYSHUTDOWN.COM

HP recommends that these site-specific SSL command procedures are utilized
to tailor the SSL installation specific to the reqirements of the system
or site. These files are located in the SSL$COM: directory.

Refer to SYS$HELP:SSL013.RELEASE_NOTES for more information.

The SSL product release notes contain up to date information regarding
bug fixes, known problems, and general installation information.
```

```
%PCSIUI-I-COMPWERR, operation completed after explicit continuation from errors
$
```

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 SSL` 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.

1.5 Postinstallation Tasks

After the installation is complete, perform the steps in one of the following sections, depending on the installation method you used.

1.5.1 After Automatic Installation of HP SSL During OpenVMS Installation or Upgrade

1. If you previously installed HP SSL, the existing file SSL$STARTUP.COM has been renamed SSL$STARTUP.COM_OLD. If you made changes to that file, manually incorporate your changes from SSL$STARTUP.COM_OLD into the new SSL$STARTUP.COM that was installed with Version 1.3.
2. 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
   ```

3. 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 certificate requests. For information about the Certificate Tool, see Chapter 3.

NOTE Beginning in OpenVMS Version 8.3, HP SSL for OpenVMS is automatically started when OpenVMS is started. The HP SSL startup file SSL$STARTUP.COM has been added to the OpenVMS command procedure VMS$LPBEGIN-050_STARTUP.COM. Startup of HP SSL Version 1.3 is required because other OpenVMS components, such as iCAP and Encrypt, are dependent on HP SSL.

1.5.2 After Download and Installation of HP SSL from Web Site

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

   ```
   $ @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 HP SSL immediately. If you installed HP SSL to a common system disk in a cluster, execute this command on each node in the cluster.

   ```
   $ @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
   ```

1.6 HP SSL Directory Structure

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

[SSL] - Top-level directory created by default in SYS$SYSDEVICE:[VMS$COMMON].
One of the following three directories:
  [SSL.ALPHA_EXE] - Contains images for the Alpha platform.
  [SSL.IA64_EXE] - Contains images for the I64 platform.
  [SSL.VAX_EXE] - Contains images for the VAX 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).
[SYS$STARTUP] - Contains startup and shutdown templates and files.
[SYSHLP] - Contains release notes.
[SYSHLP.EXAMPLES.SSL] - Contains SSL crypto and secure session examples.
[SYSLIB] - Contains SSL shareable image files.
[SYSTEST] - Contains SSL$IVP.COM test files.

Note that the HP 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 5.

1.7 Building an HP SSL Application

HP 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.

1.7.1 Building an Application Using 64-Bit APIs

To build (compile and link) an example 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
```

1.7.2 Building an Application Using 32-Bit APIs

To build (compile and link) an example 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
```

1.8 Release Notes

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

1.8.1 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 HP SSL is in compliance with all national and international laws that apply to you.

1.8.2 HP SSL APIs Not Backward Compatible

HP cannot guarantee the backward compatibility of HP SSL for OpenVMS until the release of HP SSL for OpenVMS that is based on OpenSSL 1.0.0 from The Open Group.

The HP SSL Version 1.3 for OpenVMS code is based on the 0.9.7e baselevel of OpenSSL. Any OpenSSL API, data structure, header file, command, and so on might be changed in a future version of OpenSSL.

NOTE The HP SSL shareable images use EQUAL 1,0 which means that applications will have to relink when the idents on the shareable images have changed, as they have in HP SSL Version 1.3.

If you were running a version of HP SSL prior to Version 1.2, you must recompile and relink your code after you upgrade to Version 1.3. You must relink your code if you see the following error:

```
$ run ssl_test
%DCL-W-ACTIMAGE, error activating image SSL$LIBSSL_SHR32
-CLI-E-IMGNAME, image file DWLLNG$DKA500:[SYS0.SYSCOMMON.][SYSLIB]SSL$LIBSSL_SHR32.EXE
-SYSTEM-F-SHRIDMISMAT, ident mismatch with shareable image
$
```

1.8.3 Changes to APIs in OpenSSL 0.9.7e

A number of APIs have been changed in HP SSL Version 1.3. See Appendix B for a list of new and changed APIs.

1.8.4 Preserve Configuration Files Before Manually Uninstalling HP SSL

Preserving configuration files is not necessary when you perform a regular upgrade or reinstallation of HP SSL using the PRODUCT INSTALL command.

Using the PRODUCT REMOVE command to manually uninstall HP SSL is not recommended (see the following note). However, if you made any modifications to the HP SSL configuration files, preserve the files by backing up these files to a different disk and directory before you enter the PRODUCT REMOVE command that removes the HP SSL kit. Otherwise, any changes you made to OPENSSL-VMS.CNF and OPENSSL.CNF will be lost. When you have completed the Version 1.3 installation, move the saved items back into the HP SSL directory structure.

1.8.5 Warning Against Uninstalling HP SSL from OpenVMS Version 8.3 or Higher Using the PRODUCT REMOVE Command

The POLYCENTER Software Installation utility command PRODUCT REMOVE is not supported for HP SSL on OpenVMS Version 8.3 or higher, even though there is an apparent option to remove HP SSL. HP SSL is installed together with the operating system and is tightly bound with it. An attempt to remove it from Version 8.3 or higher would not work cleanly and could create other undesirable side effects.

If you ignore the warning and continue to remove HP SSL, HP strongly recommends that you use PRODUCT INSTALL to install the HP SSL Version 1.3 PCSI kit as soon as possible. An attempt to remove HP SSL results in the following message:

```
%PCSI-E-HRDREF, product HP AXPVMS SSL V1.3-xxx is referenced by DEC AXPVMS OPENVMS
V8.3-xxx

    The two products listed above are tightly bound by a software dependency.
    If you override the recommendation to terminate the operation, the
    referenced product will be removed, but the referencing product will have
    an unsatisfied software dependency and may no longer function correctly.
    Please review the referencing product's documentation on requirements.

    Answer YES to the following question to terminate the PRODUCT command.
    However, if you are sure you want to remove the referenced product then
```

```
    answer NO to continue the operation.

Terminating is strongly recommended.  Do you want to terminate? [YES]
```

1.8.6 SSL$DEFINE_ROOT.COM Removed From SSL$STARTUP.COM

Beginning in HP SSL Version 1.3, SSL is installed on the system disk only. To reflect this change, the command procedure SSL$DEFINE_ROOT.COM has been removed from SSL$STARTUP.COM. (SSL$DEFINE_ROOT.COM was included in HP SSL Version 1.2 to define the logical SSL$ROOT. In Version 1.2, it was possible to install HP SSL to locations other than the system disk.)

The logical name SSL$ROOT is now defined in SSL$STARTUP.COM, and points to SYS$SYSDEVICE:[VMS$COMMON.SSL.].

1.8.7 SSL$STARTUP.TEMPLATE Removed From HP SSL Version 1.3

HP SSL Version 1.3 no longer contains SSL$STARTUP.TEMPLATE. Before overwriting the file, HP SSL copies your existing SSL$STARTUP.COM file to SSL$STARTUP.COM_OLD to preserve any changes that you may have made to SSL$STARTUP.COM in the past.

If you are upgrading from a previous version of HP SSL, after the installation is complete compare your SSL$STARTUP.COM_OLD file and the new SSL$STARTUP.COM file, and add any modifications you made to the new file. (Version 1.3 continues to provide the configuration template files OPENSSL.CNF_TEMPLATE and OPENSSL-VMS.CNF_TEMPLATE. See the following note for more information.)

Use SSL$COM:SSL$SYSTARTUP.COM to make additions or changes to the startup of HP SSL. SSL$COM:SSL$SYSTARTUP.COM is executed from SSL$STARTUP.COM. SSL$STARTUP.COM has been added to the OpenVMS command procedure VMS$LPBEGIN-050_STARTUP.COM so that SSL is started when OpenVMS is started.

1.8.8 Configuration Command Procedure Template Files

The configuration files included in the HP SSL kit are named OPENSSL.CNF_TEMPLATE and OPENSSL-VMS.CNF_TEMPLATE. This prevents PCSI from overwriting the .CNF files, and allows you to preserve any modifications you made to OPENSSL.CNF and OPENSSL-VMS.CNF if you installed a previous release of HP SSL for OpenVMS.

If you are upgrading from a previous version of HP SSL, after you install the HP SSL kit, compare the new .CNF_TEMPLATE files with your existing .CNF files and add any new information as required.

If you did not previously install an HP SSL for OpenVMS kit, both the .CNF_TEMPLATE and .CNF files are provided.

1.8.9 HP SSL Requirement to Install on System Disk

The option to install to a location other than the system disk is no longer available beginning in HP SSL Version 1.3. HP SSL is installed on the system disk automatically when you install or upgrade to OpenVMS Version 8.3. If you download HP SSL Version 1.3 from the web site and install it as a layered product, it too must be installed on the system disk.

1.8.10 Shut Down HP SSL Before Installing on Common System Disk

Before installing HP SSL to a common system disk in a cluster, you must first shut down HP SSL by entering the following command **on each node** in the cluster:

```
$ @SYS$STARTUP:SSL$SHUTDOWN
```

Shutting down HP SSL deassigns logical names and removes installed shareable images that may interfere with the installation.

After the installation is complete, start HP SSL by entering the following command on each node in the cluster:

```
$ @SYS$STARTUP:SSL$STARTUP
```

Note: If you are installing on a common cluster disk and not a common system disk, omit the SYS$STARTUP logical and specify the specific startup directory in the shutdown and startup commands. For example:

```
$ @device:[directory.SYS$STARTUP]SSL$SHUTDOWN
$ @device:[directory.SYS$STARTUP]SSL$STARTUP
```

1.8.11 OpenSSL Version Command Displays HP SSL for OpenVMS Version

Beginning with HP SSL Version 1.2, the OpenSSL command line utility command VERSION now includes the HP SSL for OpenVMS version. The OpenSSL VERSION command displays output similar to the following:

```
$ OPENSSL VERSION
OpenSSL 0.9.7e 25 Oct 2004
SSL for OpenVMS V1.3 May 26 2006
```

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

HP 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 HP SSL Application.

1.8.13 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 HP 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
```

1.8.14 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, which could cause database corruption.

1.8.15 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.

1.8.16 Enhancements to the HP SSL Example Programs

Beginning with HP SSL Version 1.2, several enhancements and changes were made to the HP SSL example programs located in SYS$COMMON:[SYSHLP.EXAMPLES.SSL]. These include new examples (for example, using HP SSL with QIO, AES encryption, and SHA1DIGEST) and additional common callbacks and routines to SSL_EXAMPLES.H includes file. Extra calls to free routines have been removed from the examples along with general code clean up. For more information about the example programs, see Chapter 5.

1.8.17 SSL$EXAMPLES Logical Name

The SSL$EXAMPLES logical name has been added to the SSL$STARTUP.TEMPLATE command procedure. This logical points to the directory SYS$COMMON:[SYSHLP.EXAMPLES.SSL].

1.8.18 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, HP SSL for OpenVMS only accepts the *${var}* format.

1.8.19 IDEA and RC5 Symmetric Cipher Algorithms Not Supported

The IDEA and RC5 symmetric cipher algorithms are not available in HP 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.com`

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.

1.8.20 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.

1.8.21 Documentation from the OpenSSL Web Site

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.

1.8.22 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.

1.8.23 Known Problem: Certificate Verification with OpenVMS File Specifications

OpenSSL is unable to properly parse OpenVMS file specifications when they are passed in as CApath directories. If you try to do this, OpenSSL returns the following error:

```
unable to get local issuer certificate
```

To work around this problem, define a logical that points to the OpenVMS directory, as follows:

```
$ define vms_cert_dir dka300:[ssl.certificates]
$ openssl verify "-CApath" vms_cert_dir -purpose any example.crt
```

1.8.24 Known Problem: 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));
```

1.8.25 Known Problem: Server Hang in HP SSL Session Reuse Example Program

In HP SSL Version 1.1-B and higher, a server hang problem may occur when you are running one of the HP SSL session reuse example programs. The server hang occurs when a VAX system acts as a client and the server is an Alpha or I64 system in this mixed architecture, client-server test.

When the client SSL$CLI_SESS_REUSE.EXE program is run on a VAX system, and the server SSL$SERV_SESS_REUSE.EXE program is run on an Alpha or I64 system, the server appears to hang waiting for further session reconnections, because the loop counts differ. In fact, the VAX client has finished and closed the connection. There is no problem when the client server roles are reversed, or if the same system acts as both client and server.

1.8.26 Known Problem: 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 or higher.

- 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
```

1.8.27 Problem Corrected: Possible Errors Using PRODUCT REMOVE

In HP SSL Version 1.2, when you used the PCSI REMOVE SSL command to remove previous versions of HP SSL, certain DCL symbols were not set up properly. This would result in various file not found errors.

This problem has been corrected in HP SSL Version 1.3.

1.8.28 Problem Corrected: Error Running OpenSSL Command Line Utility on ODS-5 Disks

In previous versions of HP SSL, an invalid command error was displayed when you tried to run OpenSSL commands on an ODS-5 disk with the following parsing logicals set:

```
$ SET PROCESS/PARSE=EXTENDED
$ DEFINE DECC$ARGV_PARSE_STYLE ENABLE
```

This problem has been corrected beginning in HP SSL Version 1.2. OpenSSL commands now work on both ODS-2 and ODS-5 disks, regardless of the parse settings.

1.8.29 Problem Corrected: Attempt to Encrypt within SMIME Subutility Caused Access Violation

In versions of HP SSL earlier than Version 1.2, if you entered an OpenSSL SMIME command, an access violation was returned. For example:

```
$ openssl smime -encrypt -in in.txt ssl$certs:server.pem

%SYSTEM-F-ACCVIO, access violation, reason mask=00, virtual
address=FFFFFFFFF00D2B10,
PC=000000000017DD0C, PS=0000001B
  Improperly handled condition, image exit forced.
```

This problem was corrected in OpenSSL 0.9.7d, and has been included beginning in HP SSL Version 1.2.

1.8.30 Problem Corrected: Race Condition When CRLs are Checked in a Multithreaded Environment

In versions of HP SSL earlier than Version 1.2, a race condition would occur when CRLs were checked in a multithreaded environment. This would happen because of the reordering of the revoked entries during signature checking and serial number lookup.

In OpenSSL 0.9.7e and HP SSL Version 1.2 and higher, the encoding is cached and the serial number sort is performed under a lock.

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 *HP 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 *HP 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

2.1 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.

2.2 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.

2.3 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.

2.4 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.

2.5 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.

2.6 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

3.1 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 or 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, which could cause database corruption. 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

VM-0868A-AI

3.2 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 3

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

3.3 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)

3.4 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

3.4.1 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]
```

3.5 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

3.6 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
   - Require Unique Subject Names

     If you accept the default or answer YES, then certificates must have unique subject names. If you answer NO, then certificates can have duplicate subject names, and are distinguished from one another by the serial number that is assigned to them. Answering NO allows you to have two certificates with the same subject name in the database. This makes it easier to issue new certificates when the old certificates are about to expire.

NOTE The UNIQUE_SUBJECT variable in the OPENSSL-VMS.CNF configuration file is set to YES or NO, depending on the answer to the Require Unique Subject Names question. After a CA and its database is created, the UNIQUE_SUBJECT variable should not be changed. If at a later time you want to change the setting, you must recreate the entire database.

- 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

3.7 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.)

3.7.1 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).

3.7.2 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.

3.7.3 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
```

3.8 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

3.9 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).

3.10 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.

3.11 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.
   ```

3.12 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 HP 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 HP SSL application program in C or C++.

This chapter provides the following information:

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

4.1 HP 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.

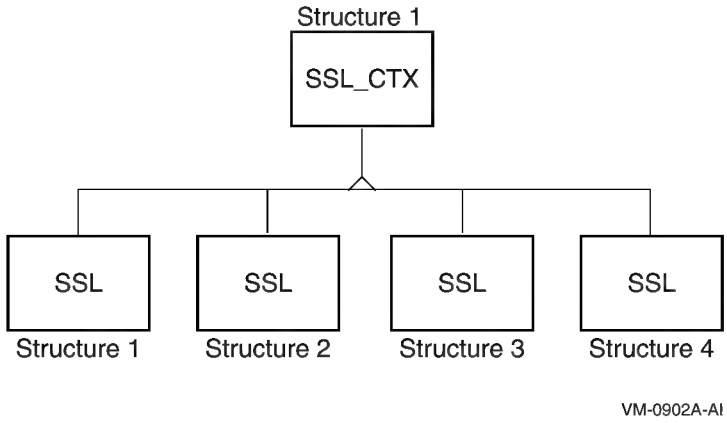
Table 4-1 shows the APIs commonly used for creating and deallocating data structures.

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



4.1.1 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.

4.1.2 SSL Structure

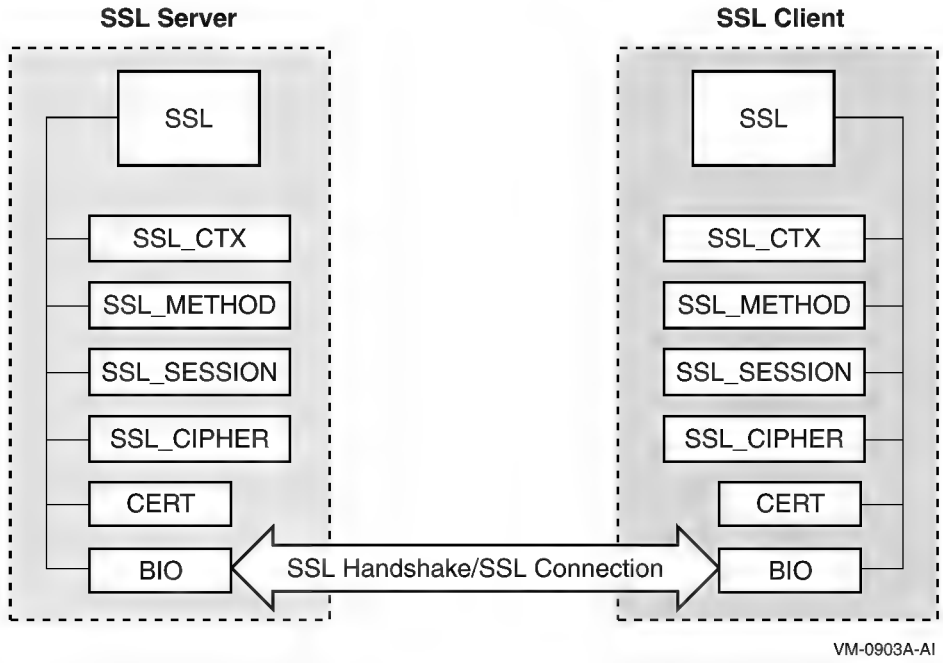
An SSL structure is created for every SSL connection in the SSL client or server program. You create the SSL structure after creating and configuring the SSL_CTX structure because the SSL structure inherits default values from the SSL_CTX structure. The inheritance of the default values enables the SSL 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



4.1.3 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.

4.1.4 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.

4.1.5 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.

4.1.6 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`.

4.2 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.

4.2.1 Configuring Certificates in the SSL Client and Server

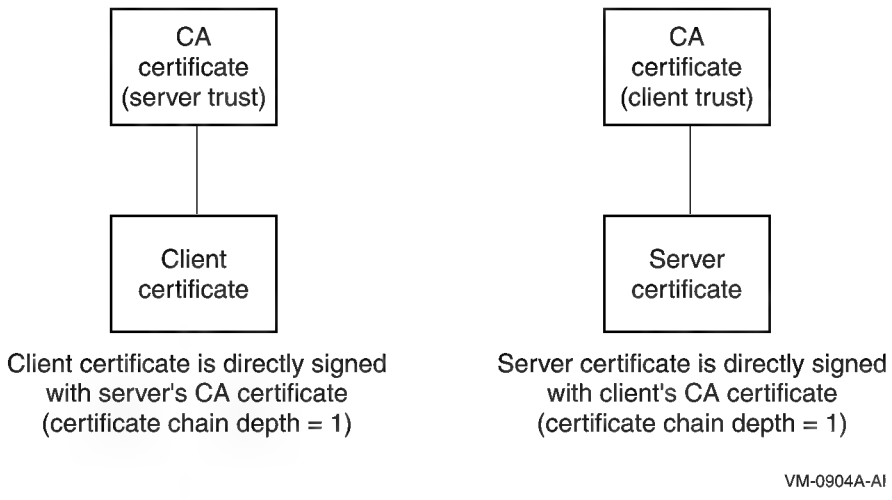
SSL client and server applications might require four certificates:

- 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

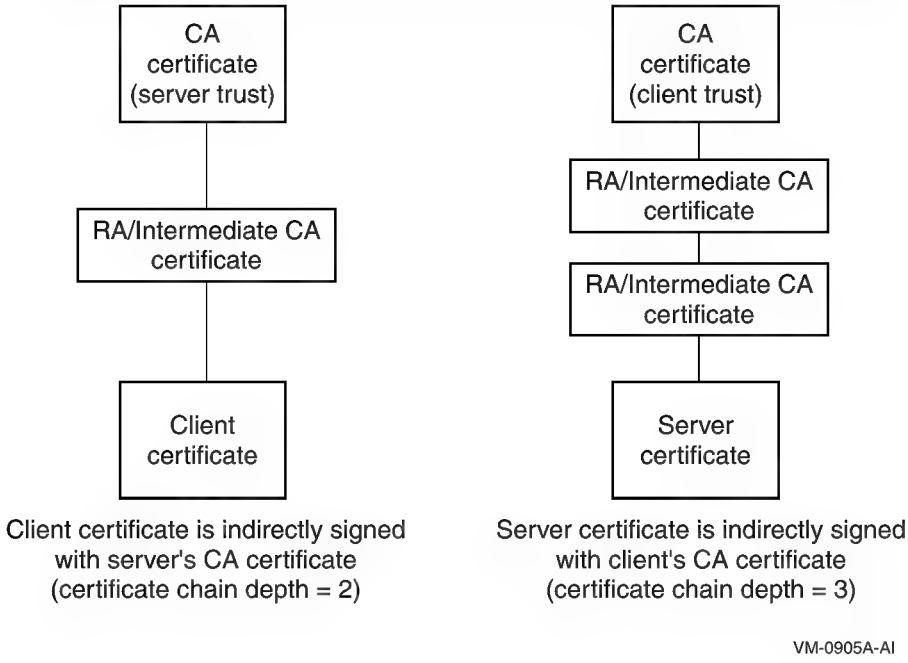
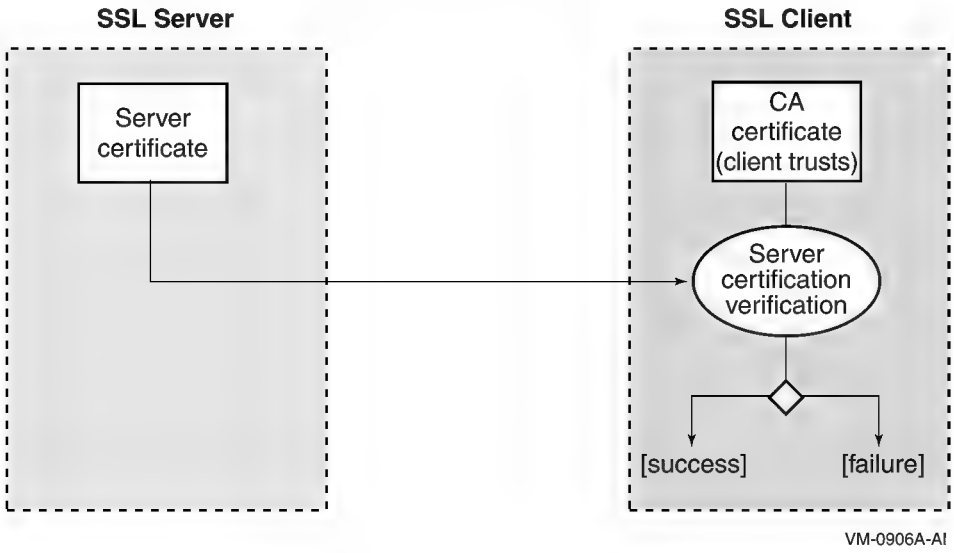


Figure 4-5 depicts the most common deployment of certificates. This deployment is often used when establishing SSL connections between web browsers and a web server. As part of its initialization, the SSL server loads a certificate (server certificate) signed by a CA. This CA is trusted by the SSL clients. 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.

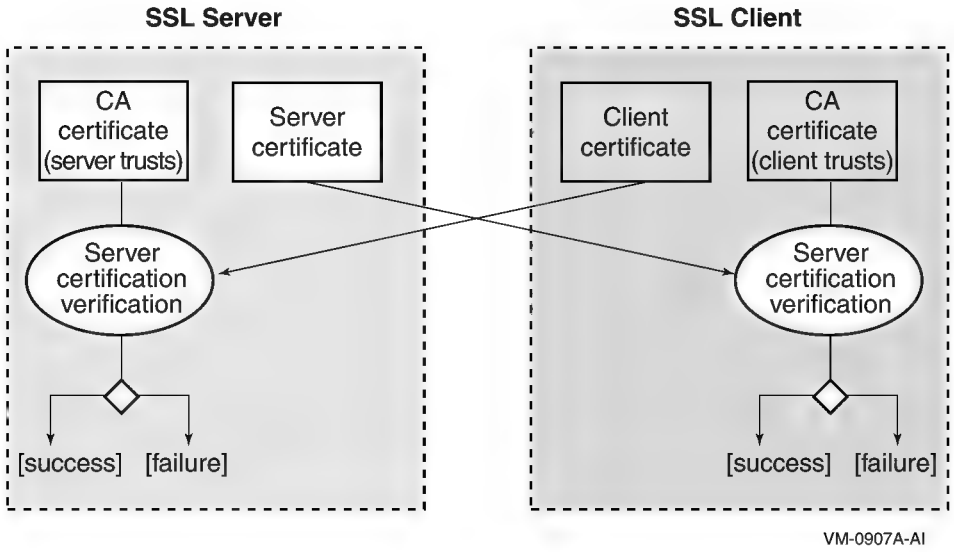
Figure 4-5 Certificates on SSL Client and Server (Case 1)



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.

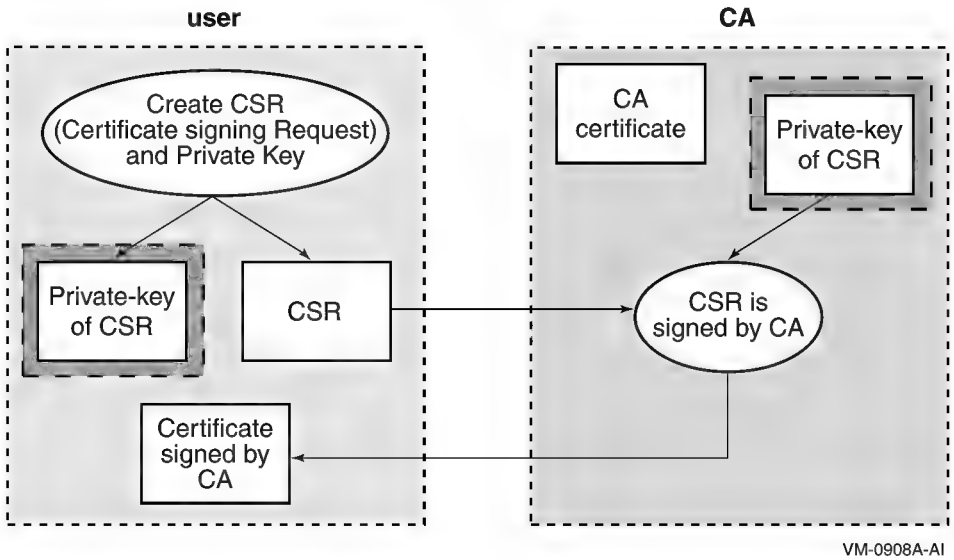
Figure 4-6 Certificates on SSL Client and Server (Case 2)



4.2.2 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.

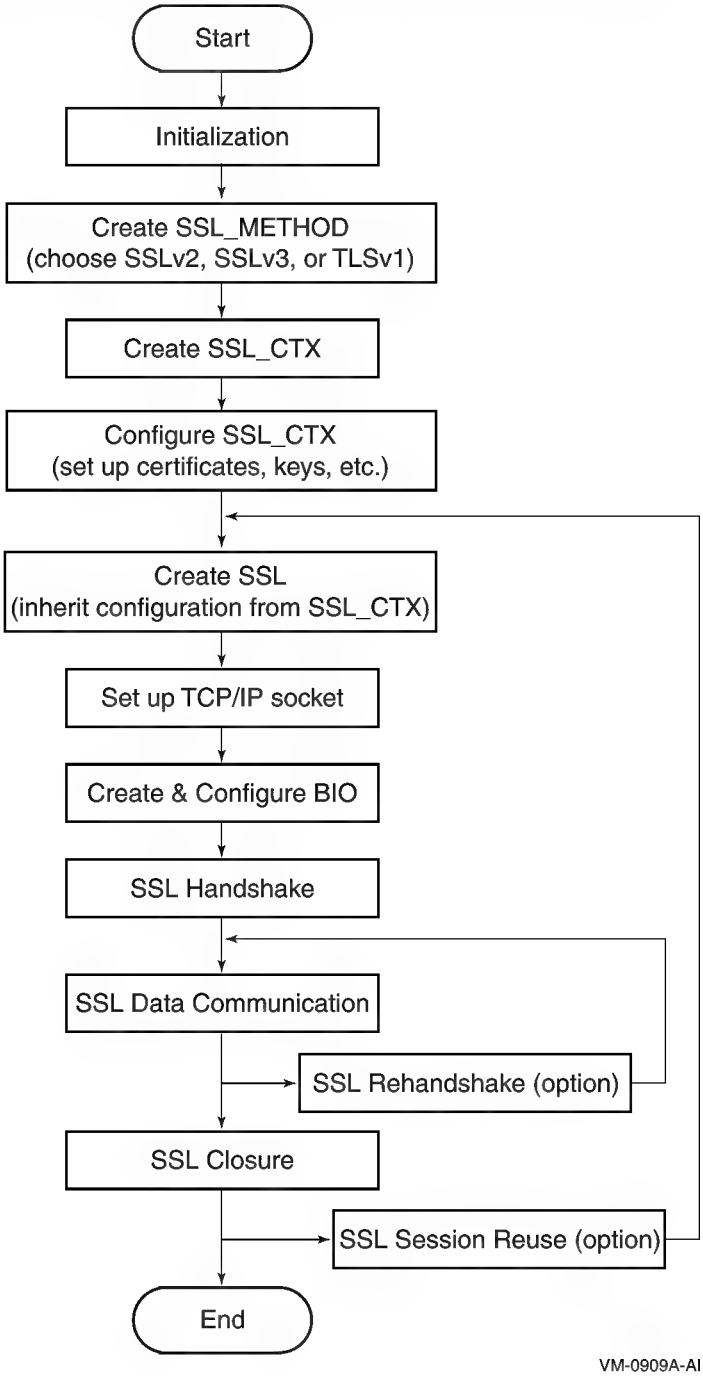
Many countries have national policies regarding encryption. Using and exporting strong encryption algorithms and keys might be affected by these policies. Also, some organizations might have policies that disallow their employees using strong encryption.

4.3 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



4.3.1 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.

4.3.2 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);
```

4.3.3 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)

4.3.3.1 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.

4.3.3.2 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()`

4.3.3.3 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.

4.3.3.4 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-);
```

4.3.3.5 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);
```

4.3.3.6 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);
        }
```

4.3.4 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.

4.3.5 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.

4.3.5.1 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");
```

4.3.5.2 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");
```

4.3.5.3 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);
```

4.3.6 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).

4.3.7 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.

4.3.7.1 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);
```

4.3.7.2 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);
```

4.3.7.3 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);
```

4.3.7.4 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);
```

4.3.8 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.

4.3.8.1 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));
```

4.3.8.2 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);
```

4.3.8.3 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[] */
```

4.3.9 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.

4.3.10 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()`.

4.3.11 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.

4.3.11.1 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.

4.3.11.2 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.

4.3.12 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 Example 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 table containing the names and descriptions of the example programs included in the kit, the template file SSL$EXAMPLES_SETUP.TEMPLATE, which sets up the certificates and keys so you can run the example programs, and the program listings of two simple example programs.

5.1 Example 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 5-1.

Table 5-1 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$AES.C	Uses SSL Advanced Encryption Standard (AES) 256-bit key encryption application program interface calls to encrypt 79 characters of data, writing the encrypted data to file, then decrypting the data and writing the plain text to a file.
SSL$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.

Table 5-1 HP SSL Example Programs (Continued)

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.
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.
SSL$SHA1_MD5.C	Uses SSL crypto library SHA1 or MD5 message disgest EVP application program interface calls to perform a one way hash on the input buffer data input1 and input2. The resulting hashed output in digest is then printed in hex format to the terminal.
SSL$TCP_CLIENT_QIO_SSL.C and SSL$TCP_SERVER_QIO_SSL.C	Demonstrate a TCP/IP IPv4 client and server using OpenVMS QIO system services to handle network I/O operations with SSL to secure the data with encryption.

5.2 Template for 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 included with HP SSL. 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
$!    SSL$CERT: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$CERT:SERVER_CA.CRT SSL$CERT: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.
$!
$!
$! Check to make sure SSL has been started, so
$! we can use the logicals that it defines.
$!
$! $ if f$trnlnm("SSL$ROOT") .eqs. ""
$! $ then
$! $    write sys$output "SSL needs to be started.  Execute SYS$STARTUP:SSL$STARTUP,"
$! $    write sys$output "then try this procedure again."
$! $ endif
$!
$! Check to make sure SSL$UTILS has been executed, so
```

```
$! we can use the foreign commands that it sets up.
$!
$! $ if f$type(OPENSSL) .eqs. ""
$! $ then
$! $    @SSL$COM:SSL$UTILS
$! $ endif
$!
$! Check to make sure the SERIAL and INDEX files exist.
$! If they don't, create them.
$!
$! $ if f$search ("SSL$ROOT:[DEMOCA]SERIAL.TXT") .eqs. ""
$! $ then
$! $   CREATE SSL$ROOT:[DEMOCA]SERIAL.TXT
$! 01
$! $ endif
$!
$! $ if f$search ("SSL$ROOT:[DEMOCA]INDEX.TXT") .eqs. ""
$! $ then
$! $   CREATE SSL$ROOT:[DEMOCA]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 -
$! -keyout ssl$key:server_ca.key -out ssl$certs:server_ca.crt
$! yourpassword
$! yourpassword
$! yourcountry
$! yourstate
$! yourcity
$! yourcompany
$! yourdepartment
$! your Certificate Authority certificate $! firstname.lastname@yourcompany.com
$!
$! Copy the server_ca.* to client_ca.* so that the CA can $! be loaded on each side.
$!
$! $ copy ssl$key:server_ca.key ssl$key:client_ca.key
$! $ copy ssl$certs:server_ca.crt ssl$certs:client_ca.crt
$!
$! $!
$! $!
$! $! 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" -
$!     + "-keyout ssl$key:server.key -out ssl$certs: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$root:[000000]openssl-vms.cnf -cert ssl$certs:server_ca.crt
-keyfile ssl$key:server_ca.key -
$!-out ssl$certs:server.crt -infiles ssl$certs: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 -
$! -keyout ssl$key:client.key -out ssl$certs: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$root:[000000]openssl-vms.cnf -cert ssl$certs:client_ca.crt
-keyfile ssl$key:client_ca.key -
$! -out ssl$certs:client.crt -infiles ssl$certs:client.csr
$! yourpassword
$! Y
$! Y
$! $!
$! $! Let's view the CA certificate.
$! $!
$! $ openssl x509 -noout -text -in ssl$certs:server_ca.crt
$! $!
```

```
$! $!
$! $! Let's view the Server Certificate Request.
$! $!
$! $ openssl req -noout -text -in ssl$certs:server.csr
$! $!
$! $! Let's view the Server Certificate.
$! $!
$! $ openssl x509 -noout -text -in ssl$certs:server.crt
$! $!
$! $! Let's view the Client Certificate Request.
$! $!
$! $ openssl req -noout -text -in ssl$certs:client.csr
$! $!
$! $! Let's view the Client Certificate.
$! $!
$! $ openssl x509 -noout -text -in ssl$certs:client.crt
$! $!
$! $!
$! $! Lastly, move the certificates and keys to the directory
$! $! in which you are building/running the examples.
$!
$! $exit
```

5.3 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.
*

*/
/* 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:SSL$LIBCRYPTO_SHR32.EXE/SHARE
*       SYS$LIBRARY:SSL$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:SSL$LIBCRYPTO_SHR.EXE/SHARE
*       SYS$LIBRARY:SSL$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);
}
```

5.4 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.
 *
 */
/* 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:SSL$LIBCRYPTO_SHR32.EXE/SHARE
*       SYS$LIBRARY:SSL$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:SSL$LIBCRYPTO_SHR.EXE/SHARE
*       SYS$LIBRARY:SSL$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);

}
```

6 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

For reference information about the OpenSSL commands, see the OpenSSL Command Line Interface (CLI) Reference.

6.1 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`.

6.2 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

6.3 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

6.4 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

6.5 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.

6.6 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)
```

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

```
 $ 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
```

OpenSSL Command Line Interface (CLI) Reference

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

`http://www.openssl.org`

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

See Chapter 6, OpenSSL Command Line Interface, for more information about the OpenSSL commands.

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]
```

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.

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

  don't 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

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

OUTPUT

The output will typically contain lines like this:

```
  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. d=XX specifies the current depth. The depth is increased within the scope of any SET or SEQUENCE.

h1=XX gives the header length (tag and length octets) of the current type. 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 contents octets of this will contain the public key information. This can be examined 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
:E5D21E1F5C8D208EA7A2166C7FAF9F6BDF2059669C60876DDB70840F1A5AAFA59699FE471F379F1DD6A487E7D540
9AB6A88D4A9746E24B91D8CF55DB3521015460C8EDE44EE8A4189F7A7BE77D6CD3A9AF2696F486855CF58BF0EDF2B
4068058C7A947F52548DDF7E15E96B385F86422BEA9064A3EE9E1158A56E4A6F47E5897
  135:d=1  hl=2 l=   3 prim: INTEGER           :010001
```

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. 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 well handled (if at all).

ca

NAME

ca – sample minimal CA application

Synopsis

```
openssl ca [-verbose] [-config filename] [-name section] [-gencrl] [-revoke file]
[-crl_reason reason] [-crl_hold instruction] [-crl_compromise time] [-crl_CA_compromise
time] [-subj arg] [-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] [-noemailDN] [-batch] [-msie_hack] [-extensions section]
[-extfile section] [-engine id]
```

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.

CA OPTIONS

- -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 SPKAC FORMAT 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 PHRASE ARGUMENTS section in *openssl* (1).

- -verbose

  this prints extra details about the operations being performed.

- -notext

  don't output the text form of a certificate to the output file.

- -startdate date

  this allows the start date to be explicitly set. The format of the date is YYMMDDHHMMSSZ (the same as an ASN1 UTCTime structure).

- -enddate date

  this allows the expiry 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 alg

  the message digest to use. Possible values include md5, sha1 and mdc2. This option also applies to CRLs.

- -policy arg

  this option 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. Check out the POLICY FORMAT section for more information.

- -msie_hack

  this is 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.

- -noemailDN

The DN of a certificate can contain the EMAIL field if present in the request DN, however it is good policy just having the e-mail set into the altName extension of the certificate. When this option is set the EMAIL field is removed from the certificate' subject and set only in the, eventually present, extensions. The email_in_dn keyword can be used in the configuration file to enable this behaviour.

- -batch

  this 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 (defaults to x509_extensions unless the -extfile option is used). 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.

- -extfile file

  an additional configuration file to read certificate extensions from (using the default section unless the -extensions option is also used).

- -engine id

  specifying an engine (by it's unique id string) will cause req to attempt to obtain a functional reference to the specified engine, thus initialising it if needed. The engine will then be set as the default for all available algorithms.

CRL OPTIONS

- -gencrl

  this option 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 nextUpdate field.

- -crlhours num

  the number of hours before the next CRL is due.

- -revoke filename

  a filename containing a certificate to revoke.

- -crl_reason reason

  revocation reason, where reason is one of: unspecified, keyCompromise, CACompromise, affiliationChanged, superseded , cessationOfOperation, certificateHold or removeFromCRL . The matching of reason is case insensitive. Setting any revocation reason will make the CRL v2.

  In practive removeFromCRL is not particularly useful because it is only used in delta CRLs which are not currently implemented.

- -crl_hold instruction

  This sets the CRL revocation reason code to certificateHold and the hold instruction to instruction which must be an OID. Although any OID can be used only holdInstructionNone (the use of which is discouraged by RFC2459) holdInstructionCallIssuer or holdInstructionReject will normally be used.

- -crl_compromise time

  This sets the revocation reason to keyCompromise and the compromise time to time. time should be in GeneralizedTime format; that is, YYYYMMDDHHMMSSZ.

- -crl_CA_compromise time

  This is the same as crl_compromise except the revocation reason is set to CACompromise .

- -subj arg

  supersedes subject name given in the request. The arg must be formatted as */type0=value0/type1=value1/type2=...*, characters may be escaped by \ (backslash), no spaces are skipped.

- -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 (for example Netscape) can't 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) used.

- oid_file

  This 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

  This 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 -cert. It gives the file containing the CA certificate. Mandatory.

- private_key

  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* (3)).

- 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.

- serial

  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 -extensions.

- crl_extensions

  the same as -crlexts.

- preserve

  the same as -preserveDN

- email_in_dn

  the same as -noemailDN. If you want the EMAIL field to be removed from the DN of the certificate simply set this to 'no'. If not present the default is to allow for the EMAIL filed in the certificate's DN.

- msie_hack

  the same as -msie_hack

- policy

  the same as -policy. Mandatory. See the POLICY FORMAT section for more information.

- nameopt, certopt

  these options allow the format used to display the certificate details when asking the user to confirm signing. All the options supported by the x509 utilities -nameopt and -certopt switches can be used here, except the no_signame and no_sigdump are permanently set and cannot be disabled (this is because the certificate signature cannot be displayed because the certificate has not been signed at this point).

  For convenience the values ca_default are accepted by both to produce a reasonable output.

If neither option is present the format used in earlier versions of OpenSSL is used. Use of the old format is strongly discouraged because it only displays fields mentioned in the policy section, mishandles multicharacter string types and does not display extensions.

- copy_extensions

  determines how extensions in certificate requests should be handled. If set to none or this option is not present then extensions are ignored and not copied to the certificate. If set to copy then any extensions present in the request that are not already present are copied to the certificate. If set to copyall then all extensions in the request are copied to the certificate: if the extension is already present in the certificate it is deleted first. See the WARNINGS section before using this option.

  The main use of this option is to allow a certificate request to supply values for certain extensions such as subjectAltName.

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 behaviour.

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 '.'.

EXAMPLES

Note: 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
email_in_dn     = no                    # Don't add the email into cert DN

nameopt= ca_default# Subject name display option
certopt= ca_default# Certificate display option
copy_extensions = none# Don't copy extensions from request

[ policy_any ]
countryName             = supplied
stateOrProvinceName     = optional
organizationName        = optional
organizationalUnitName  = optional
commonName              = supplied
emailAddress            = optional
```

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
```

ENVIRONMENT VARIABLES

OPENSSL_CONF reflects the location of master configuration file it can be overridden by the -config command line option.

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.

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.

Restrictions

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.

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. To enforce the absence of the EMAIL field within the DN, as suggested by RFCs, regardless the contents of the request' subject the -noemailDN option can be used. The behaviour should be more friendly and configurable.

Cancelling some commands by refusing to certify a certificate can create an empty file.

WARNINGS

The ca command is quirky and at times downright unfriendly.

The ca utility was originally meant as an example of how to do things in a CA. It was not supposed to be used as a full blown CA itself: 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.

The copy_extensions option should be used with caution. If care is not taken then it can be a security risk. For example if a certificate request contains a basicConstraints extension with CA:TRUE and the copy_extensions value is set to copyall and the user does not spot this when the certificate is displayed then this will hand the requestor a valid CA certificate.

This situation can be avoided by setting copy_extensions to copy and including basicConstraints with CA:FALSE in the configuration file. Then if the request contains a basicConstraints extension it will be ignored.

It is advisable to also include values for other extensions such as keyUsage to prevent a request supplying its own values.

Additional restrictions can be placed on the CA certificate itself. For example if the CA certificate has:

```
basicConstraints = CA:TRUE, pathlen:0
```

then even if a certificate is issued with CA:TRUE it will not be valid.

SEE ALSO

req (1), *spkac* (1), `x509(1)`, `CA.`*pl* (1), *config* (5)

ciphers

NAME

ciphers – SSL cipher display and cipher list tool

Synopsis

```
openssl ciphers [-v] [-ssl2] [-ssl3] [-tls1] [cipherlist]
```

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.

COMMAND OPTIONS

- -v

  verbose option. List 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. Note that 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, -?

  print 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. The format is described below.

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 moved 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

The 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.

- COMPLEMENTOFDEFAULT

  the ciphers included in ALL, but not enabled by default. Currently this is ADH. Note that this rule does not cover eNULL, which is not included by ALL (use COMPLEMENTOFALL if necessary).

- ALL

  all ciphers suites except the eNULL ciphers which must be explicitly enabled.

- COMPLEMENTOFALL

  the cipher suites not enabled by ALL, currently being eNULL.

- HIGH

  "high" 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

  ciphers 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.

- AES

  cipher suites using AES.

- 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. It should be noted, that several cipher suite names do not include the authentication used, e.g. DES-CBC3-SHA. In these cases, RSA authentication is used.

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
```

AES ciphersuites from RFC3268, extending TLS v1.0

```
TLS_RSA_WITH_AES_128_CBC_SHA            AES128-SHA
TLS_RSA_WITH_AES_256_CBC_SHA            AES256-SHA

TLS_DH_DSS_WITH_AES_128_CBC_SHA         DH-DSS-AES128-SHA
TLS_DH_DSS_WITH_AES_256_CBC_SHA         DH-DSS-AES256-SHA
TLS_DH_RSA_WITH_AES_128_CBC_SHA         DH-RSA-AES128-SHA
TLS_DH_RSA_WITH_AES_256_CBC_SHA         DH-RSA-AES256-SHA

TLS_DHE_DSS_WITH_AES_128_CBC_SHA        DHE-DSS-AES128-SHA
TLS_DHE_DSS_WITH_AES_256_CBC_SHA        DHE-DSS-AES256-SHA
TLS_DHE_RSA_WITH_AES_128_CBC_SHA        DHE-RSA-AES128-SHA
TLS_DHE_RSA_WITH_AES_256_CBC_SHA        DHE-RSA-AES256-SHA

TLS_DH_anon_WITH_AES_128_CBC_SHA        ADH-AES128-SHA
TLS_DH_anon_WITH_AES_256_CBC_SHA        ADH-AES256-SHA
```

Additional Export 1024 and other cipher suites

Note: 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
```

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'
```

Include all RC4 ciphers but leave out those without authentication:

```
openssl ciphers -v 'RC4:!COMPLEMENTOFDEFAULT'
```

Include all chiphers with RSA authentication but leave out ciphers without encryption.

```
openssl ciphers -v 'RSA:!COMPLEMENTOFALL'
```

SEE ALSO

s_client (1), *s_server* (1), *ssl* (3)

HISTORY

The COMPLENTOFALL and COMPLEMENTOFDEFAULT selection options were added in version 0.9.7.

config

NAME

config – OpenSSL CONF library configuration files

DESCRIPTION

The OpenSSL CONF library can be used to read configuration files. It is used for the OpenSSL master configuration file openssl.cnf and in a few other places like SPKAC files and certificate extension files for the x509 utility.

A configuration file is divided into a number of sections. Each section starts with a line [section_name] and ends when a new section is started or end of file is reached. A section name can consist of alphanumeric characters and underscores.

The first section of a configuration file is special and is referred to as the default section this is usually unnamed and is from the start of file until the first named section. When a name is being looked up it is first looked up in a named section (if any) and then the default section.

The environment is mapped onto a section called ENV.

Comments can be included by preceding them with the # character

Each section in a configuration file consists of a number of name and value pairs of the form name=value

The name string can contain any alphanumeric characters as well as a few punctuation symbols such as . , ; and _.

The value string consists of the string following the = character until end of line with any leading and trailing white space removed.

The value string undergoes variable expansion. This can be done by including the form $var or ${var}: this will substitute the value of the named variable in the current section. It is also possible to substitute a value from another section using the syntax $section::name or ${section::name}. By using the form $ENV::name environment variables can be substituted. It is also possible to assign values to environment variables by using the name ENV::name, this will work if the program looks up environment variables using the CONF library instead of calling getenv() directly.

It is possible to escape certain characters by using any kind of quote or the \ character. By making the last character of a line a \ a value string can be spread across multiple lines. In addition the sequences \n, \r, \b and \t are recognized.

NOTES

If a configuration file attempts to expand a variable that doesn't exist then an error is flagged and the file will not load. This can happen if an attempt is made to expand an environment variable that doesn't exist. For example the default OpenSSL master configuration file used the value of HOME which may not be defined on non UNIX systems.

This can be worked around by including a default section to provide a default value: then if the environment lookup fails the default value will be used instead. For this to work properly the default value must be defined earlier in the configuration file than the expansion. See the EXAMPLES section for an example of how to do this.

If the same variable exists in the same section then all but the last value will be silently ignored. In certain circumstances such as with DNs the same field may occur multiple times. This is usually worked around by ignoring any characters before an initial . e.g.

```
1.OU="My first OU"
2.OU="My Second OU"
```

EXAMPLES

Here is a sample configuration file using some of the features mentioned above.

```
# This is the default section.

HOME=/temp
RANDFILE= ${ENV::HOME}/.rnd
configdir=$ENV::HOME/config

[ section_one ]

# We are now in section one.

# Quotes permit leading and trailing whitespace
any = " any variable name "

other = A string that can \
cover several lines \
by including \\ characters

message = Hello World\n

[ section_two ]

greeting = $section_one::message
```

This next example shows how to expand environment variables safely.

Suppose you want a variable called tmpfile to refer to a temporary filename. The directory it is placed in can determined by the the TEMP or TMP environment variables but they may not be set to any value at all. If you just include the environment variable names and the variable doesn't exist then this will cause an error when an attempt is made to load the configuration file. By making use of the default section both values can be looked up with TEMP taking priority and /tmp used if neither is defined:

```
TMP=/tmp
# The above value is used if TMP isn't in the environment
TEMP=$ENV::TMP
# The above value is used if TEMP isn't in the environment
tmpfile=${ENV::TEMP}/tmp.filename
```

Restrictions

Currently there is no way to include characters using the octal \nnn form. Strings are all null terminated so nulls cannot form part of the value.

The escaping isn't quite right: if you want to use sequences like \n you can't use any quote escaping on the same line.

Files are loaded in a single pass. This means that an variable expansion will only work if the variables referenced are defined earlier in the file.

SEE ALSO

`x509(1)`, *req* (1), *ca* (1)

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]
```

DESCRIPTION

The crl command processes CRL files in DER or PEM format.

COMMAND OPTIONS

- -inform DER | PEM

  This specifies the 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

  This specifies the output format, the options have the same meaning as the -inform option.

- -in filename

  This 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

  print out the CRL in text form.

- -noout

  don't output the encoded version of the CRL.

- -hash

  output a hash of the issuer name. This can be use to lookup CRLs in a directory by issuer name.

- -issuer

  output the issuer name.

- -lastupdate

  output the lastUpdate field.

- -nextupdate

  output the nextUpdate field.

- -CAfile file

  verify the signature on a CRL by looking up the issuing certificate in file

- -CApath dir

verify 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.

NOTES

The PEM CRL format uses the header and footer lines:

```
-----BEGIN X509 CRL-----
-----END X509 CRL-----
```

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
```

Restrictions

Ideally it should be possible to create a CRL using appropriate options and files too.

SEE ALSO

`crl2pkcs7(1)`, *ca* (1), `x509(1)`

crl2pkcs7

NAME

crl2pkcs7 – Create a PKCS#7 structure from a CRL and certificates

Synopsis

```
openssl crl2pkcs7 [-inform PEM|DER] [-outform PEM|DER] [-in filename] [-out filename]
[-certfile filename] [-nocrl]
```

DESCRIPTION

The crl2pkcs7 command takes an optional CRL and one or more certificates and converts them into a PKCS#7 degenerate "certificates only" structure.

COMMAND OPTIONS

- -inform DER | PEM

  This 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

  This specifies the PKCS#7 structure output format. DER format is DER encoded PKCS#7 structure. 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.

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
```

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.

SEE ALSO

`pkcs7(1)`

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] [file...] [md5|md4|md2|sha1|sha|mdc2|ripemd160] [-c] [-d] [file...]
```

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.

OPTIONS

- -c

  print out the digest in two digit groups separated by colons, only relevant if hex format output is used.

- -d

  print out BIO debugging information.

- -hex

  digest is to be output as a hex dump. This is the default case for a "normal" digest as opposed to a digital signature.

- -binary

  output the digest or signature in binary form.

- -out filename

  filename to output to, or standard output by default.

- -sign filename

  digitally sign the digest using the private key in "filename".

- -verify filename

  verify the signature using the the public key in "filename". The output is either "Verification OK" or "Verification Failure".

- -prverify filename

  verify the signature using the 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* (3)). Multiple files can be specified separated by a OS-dependent character. The separator is ; for MS-Windows, , for OpenVMS, and : for all others.

- file...

  file or files to digest. If no files are specified then standard input is used.

NOTES

The digest of choice for all new applications is SHA1. Other digests are however 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.

dhparam

NAME

dhparam – DH parameter manipulation and generation,

Synopsis

`openssl dhparam [-inform DER|PEM] [-outform DER|PEM] [-in` *filename*`] [-out` *filename*`] [-dsaparam] [-noout] [-text] [-C] [-2] [-5] [-rand` *file(s)*`] [-engine id] [`*numbits*`]`

DESCRIPTION

This command is used to manipulate DH parameter files.

OPTIONS

- -inform DER | PEM

  This specifies the input format. The DER option uses an ASN1 DER encoded form compatible with the PKCS#3 DHparameter structure. The PEM form is the default format: it consists of the DER format base64 encoded with additional header and footer lines.

- -outform DER | PEM

  This specifies the output format, the options have the same meaning as the -inform option.

- -in *filename*

  This specifies the input filename to read parameters from or standard input if this option is not specified.

- -out *filename*

  This 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.

- -dsaparam

  If this option is used, DSA rather than DH parameters are read or created; they are converted to DH format. Otherwise, "strong" primes (such that (p-1)/2 is also prime) will be used for DH parameter generation.

  DH parameter generation with the -dsaparam option is much faster, and the recommended exponent length is shorter, which makes DH key exchange more efficient. Beware that with such DSA-style DH parameters, a fresh DH key should be created for each use to avoid small-subgroup attacks that may be possible otherwise.

- -2, -5

  The generator to use, either 2 or 5. 2 is the default. If present then the input file is ignored and parameters are generated instead.

- -rand *file(s)*

  a file or files containing random data used to seed the random number generator, or an EGD socket (see *RAND_egd* (3)). Multiple files can be specified separated by a OS-dependent character. The separator is ; for MS-Windows, , for OpenVMS, and : for all others.

- *numbits*

this option specifies that a parameter set should be generated of size *numbits*. It must be the last option. If not present then a value of 512 is used. If this option is present then the input file is ignored and parameters are generated instead.

- -noout

  this option inhibits the output of the encoded version of the parameters.

- -text

  this option prints out the DH parameters in human readable form.

- -C

  this option converts the parameters into C code. The parameters can then be loaded by calling the get_dh*numbits*() function.

- -engine id

  specifying an engine (by it's unique id string) will cause req to attempt to obtain a functional reference to the specified engine, thus initialising it if needed. The engine will then be set as the default for all available algorithms.

WARNINGS

The program dhparam combines the functionality of the programs dh and gendh in previous versions of OpenSSL and SSLeay. The dh and gendh programs are retained for now but may have different purposes in future versions of OpenSSL.

NOTES

PEM format DH parameters use the header and footer lines:

```
-----BEGIN DH PARAMETERS-----
-----END DH PARAMETERS-----
```

OpenSSL currently only supports the older PKCS#3 DH, not the newer X9.42 DH.

This program manipulates DH parameters not keys.

Restrictions

There should be a way to generate and manipulate DH keys.

SEE ALSO

dsaparam (1)

HISTORY

The dhparam command was added in OpenSSL 0.9.5. The -dsaparam option was added in OpenSSL 0.9.6.

dsa

NAME

dsa – DSA key processing

Synopsis

```
openssl dsa [-inform PEM|DER] [-outform PEM|DER] [-in filename] [-passin arg] [-out
filename] [-passout arg] [-des] [-des3] [-idea] [-text] [-noout] [-modulus] [-pubin]
[-pubout] [-engine id]
```

DESCRIPTION

The dsa command processes DSA keys. They can be converted between various forms and their components printed out.

Note: This command uses the traditional SSLeay compatible format for private key encryption: newer applications should use the more secure PKCS#8 format using the pkcs8.

COMMAND OPTIONS

- -inform DER | PEM

  This specifies the input format. The DER option with a private key uses an ASN1 DER encoded form of an ASN.1 SEQUENCE consisting of the values of version (currently zero), p, q, g, the public and private key components respectively as ASN.1 INTEGERs. When used with a public key it uses a SubjectPublicKeyInfo structure: it is an error if the key is not DSA.

  The PEM form is the default format: it consists of the DER format base64 encoded with additional header and footer lines. In the case of a private key PKCS#8 format is also accepted.

- -outform DER | PEM

  This specifies the output format, the options have the same meaning as the -inform option.

- -in filename

  This specifies the input filename to read a key from or standard input if this option is not specified. If the key is encrypted a pass phrase will be prompted for.

- -passin arg

  the input file password source. For more information about the format of arg see the PASS PHRASE ARGUMENTS section in *openssl* (1).

- -out filename

  This specifies the output filename to write a key to or standard output by is not specified. If any encryption options are set then a pass phrase will be prompted for. The output filename should not be the same as the input filename.

- -passout arg

  the output file password source. For more information about the format of arg see the PASS PHRASE ARGUMENTS section in *openssl* (1).

- -des | -des3 | -idea

  These options encrypt 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 the key is written in plain text. This means that using the dsa utility to read in an encrypted key with no encryption option can be used to remove the pass phrase from a key, or by setting the encryption options it can be use to add or change the pass phrase. These options can only be used with PEM format output files.

- -text

  prints out the public, private key components and parameters.

- -noout

  this option prevents output of the encoded version of the key.

- -modulus

  this option prints out the value of the public key component of the key.

- -pubin

  by default a private key is read from the input file: with this option a public key is read instead.

- -pubout

  by default a private key is output. With this option a public key will be output instead. This option is automatically set if the input is a public key.

- -engine id

  specifying an engine (by it's unique id string) will cause req to attempt to obtain a functional reference to the specified engine, thus initialising it if needed. The engine will then be set as the default for all available algorithms.

NOTES

The PEM private key format uses the header and footer lines:

```
-----BEGIN DSA PRIVATE KEY-----
-----END DSA PRIVATE KEY-----
```

The PEM public key format uses the header and footer lines:

```
-----BEGIN PUBLIC KEY-----
-----END PUBLIC KEY-----
```

EXAMPLES

To remove the pass phrase on a DSA private key:

```
openssl dsa -in key.pem -out keyout.pem
```

To encrypt a private key using triple DES:

```
openssl dsa -in key.pem -des3 -out keyout.pem
```

To convert a private key from PEM to DER format:

```
openssl dsa -in key.pem -outform DER -out keyout.der
```

To print out the components of a private key to standard output:

```
openssl dsa -in key.pem -text -noout
```

To just output the public part of a private key:

```
openssl dsa -in key.pem -pubout -out pubkey.pem
```

SEE ALSO

dsaparam (1), *gendsa* (1), *rsa* (1), *genrsa* (1)

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 file(s)] [-genkey] [-engine id] [numbits]
```

DESCRIPTION

This command is used to manipulate or generate DSA parameter files.

OPTIONS

- -inform DER | PEM

  This 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

  This specifies the output format, the options have the same meaning as the -inform option.

- -in filename

  This 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

  This 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

  this option inhibits the output of the encoded version of the parameters.

- -text

  this option prints out the DSA parameters in human readable form.

- -C

  this option converts the parameters into C code. The parameters can then be loaded by calling the get_dsaXXX() function.

- -genkey

  this option will generate 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* (3)). Multiple files can be specified separated by a OS-dependent character. The separator is ; for MS-Windows, , for OpenVMS, and : for all others.

- numbits

  this option 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.

- -engine id

  specifying an engine (by it's unique id string) will cause req to attempt to obtain a functional reference to the specified engine, thus initialising it if needed. The engine will then be set as the default for all available algorithms.

NOTES

PEM format DSA parameters use the 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

gendsa (1), *dsa* (1), *genrsa* (1), *rsa* (1)

enc

NAME

enc – symmetric cipher routines

Synopsis

```
openssl enc -ciphername [-in filename] [-out filename] [-pass arg] [-e] [-d] [-a] [-A] [-k
password] [-kfile filename] [-K key] [-iv IV] [-p] [-P] [-bufsize number] [-nopad] [-debug]
```

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.

OPTIONS

- -in filename

  the input filename, standard input by default.

- -out filename

  the output filename, standard output by default.

- -pass arg

  the password source. For more information about the format of arg see the PASS PHRASE ARGUMENTS section in *openssl* (1).

- -salt

  use 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

  don't use a salt in the key derivation routines. This is the default for compatibility with previous versions of OpenSSL and SSLeay.

- -e

  encrypt the input data: this is the default.

- -d

  decrypt the input data.

- -a

  base64 process 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 process the data on one line.

- -k password

  the password to derive the key from. This is for compatibility with previous versions of OpenSSL. Superseded by the -pass argument.

- -kfile filename

  read the password to derive the key from the first line of filename. This is for computability with previous versions of OpenSSL. 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 additionally specified using the -iv option. When both a key and a 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.

- -iv IV

  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 explicitly be defined. When a password is being specified using one of the other options, the IV is generated from this password.

- -p

  print out the key and IV used.

- -P

  print out the key and IV used then immediately exit: don't do any encryption or decryption.

- -bufsize number

  set the buffer size for I/O

- -nopad

  disable standard block padding

- -debug

  debug the BIOs used for I/O.

NOTES

The program can be called either as openssl ciphername or openssl enc -ciphername.

A password will be prompted for 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 just use a strong block cipher in CBC mode such as bf or des3.

All the block ciphers normally 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 isn't a very good test.

If padding is disabled then the input data must be a multiple of the cipher block length.

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
```

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
```

Restrictions

The -A option when used with large files doesn't work properly.

There should be an option to allow an iteration count to be included.

The enc program only supports a fixed number of algorithms with certain parameters. So if, for example, you want to use RC2 with a 76 bit key or RC4 with an 84 bit key you can't use this program.

gendsa

NAME

gendsa – generate a DSA private key from a set of parameters

Synopsis

```
openssl gendsa [-out filename] [-des] [-des3] [-idea] [-rand file(s)] [-engine id]
[paramfile]
```

DESCRIPTION

The gendsa command generates a DSA private key from a DSA parameter file (which will be typically generated by the openssl dsaparam command).

OPTIONS

- -des | -des3 | -idea

  These options encrypt 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 file(s)

  a file or files containing random data used to seed the random number generator, or an EGD socket (see *RAND_egd* (3)). Multiple files can be specified separated by a OS-dependent character. The separator is ; for MS-Windows, , for OpenVMS, and : for all others.

- -engine id

  specifying an engine (by it's unique id string) will cause req to attempt to obtain a functional reference to the specified engine, thus initialising it if needed. The engine will then be set as the default for all available algorithms.

- paramfile

  This option 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.

NOTES

DSA key generation is little more than random number generation so it is much quicker that RSA key generation for example.

SEE ALSO

dsaparam (1), *dsa* (1), *genrsa* (1), *rsa* (1)

genrsa

NAME

genrsa – generate an RSA private key

Synopsis

```
openssl genrsa [-out filename] [-passout arg] [-des] [-des3] [-idea] [-f4] [-3] [-rand
file(s)] [-engine id] [numbits]
```

DESCRIPTION

The genrsa command generates an RSA private key.

OPTIONS

- -out filename

  the output filename. If this argument is not specified then standard output is used.

- -passout arg

  the output file password source. For more information about the format of arg see the PASS PHRASE ARGUMENTS section in *openssl* (1).

- -des | -des3 | -idea

  These options encrypt 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 a pass phrase is prompted for 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 file(s)

  a file or files containing random data used to seed the random number generator, or an EGD socket (see *RAND_egd* (3)). Multiple files can be specified separated by a OS-dependent character. The separator is ; for MS-Windows, , for OpenVMS, and : for all others.

- -engine id

  specifying an engine (by it's unique id string) will cause req to attempt to obtain a functional reference to the specified engine, thus initialising it if needed. The engine will then be set as the default for all available algorithms.

- numbits

  the size of the private key to generate in bits. This must be the last option specified. The default is 512.

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 . represents each number which has passed an initial sieve test, + 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 somewhat.

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

gendsa (1)

nseq

NAME

nseq – create or examine a netscape certificate sequence

Synopsis

```
openssl nseq [-in filename] [-out filename] [-toseq]
```

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.

COMMAND OPTIONS

- -in filename

  This 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.

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
```

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: like allowing DER or PEM input and output files and allowing multiple certificate files to be used.

ocsp

NAME

ocsp – Online Certificate Status Protocol utility

Synopsis

```
openssl ocsp [-out file] [-issuer file] [-cert file] [-serial n] [-signer file] [-signkey
file] [-sign_other file] [-no_certs] [-req_text] [-resp_text] [-text] [-reqout file]
[-respout file] [-reqin file] [-respin file] [-nonce] [-no_nonce] [-url URL] [-host host:n]
[-path] [-CApath dir] [-CAfile file] [-VAfile file] [-validity_period n] [-status_age n]
[-noverify] [-verify_other file] [-trust_other] [-no_intern] [-no_signature_verify]
[-no_cert_verify] [-no_chain] [-no_cert_checks] [-port num] [-index file] [-CA file]
[-rsigner file] [-rkey file] [-rother file] [-resp_no_certs] [-nmin n] [-ndays n]
[-resp_key_id] [-nrequest n]
```

DESCRIPTION

The Online Certificate Status Protocol (OCSP) enables applications to determine the (revocation) state of an identified certificate (RFC 2560).

The ocsp command performs many common OCSP tasks. It can be used to print out requests and responses, create requests and send queries to an OCSP responder and behave like a mini OCSP server itself.

OCSP CLIENT OPTIONS

- -out filename

  specify output filename, default is standard output.

- -issuer filename

  This specifies the current issuer certificate. This option can be used multiple times. The certificate specified in filename must be in PEM format.

- -cert filename

  Add the certificate filename to the request. The issuer certificate is taken from the previous issuer option, or an error occurs if no issuer certificate is specified.

- -serial num

  Same as the cert option except the certificate with serial number num is added to the request. The serial number is interpreted as a decimal integer unless preceded by 0x. Negative integers can also be specified by preceding the value by a - sign.

- -signer filename, -signkey filename

  Sign the OCSP request using the certificate specified in the signer option and the private key specified by the signkey option. If the signkey option is not present then the private key is read from the same file as the certificate. If neither option is specified then the OCSP request is not signed.

- -sign_other filename

  Additional certificates to include in the signed request.

- -nonce, -no_nonce

  Add an OCSP nonce extension to a request or disable OCSP nonce addition. Normally if an OCSP request is input using the respin option no nonce is added: using the nonce option will force addition of a nonce. If an OCSP request is being created (using cert and serial options) a nonce is automatically added specifying no_nonce overrides this.

- -req_text, -resp_text, -text

  print out the text form of the OCSP request, response or both respectively.

- -reqout file, -respout file

  write out the DER encoded certificate request or response to file.

- -reqin file, respin file

  read OCSP request or response file from file . These option are ignored if OCSP request or response creation is implied by other options (for example with serial , cert and host options).

- -url responder_url

  specify the responder URL. Both HTTP and HTTPS (SSL/TLS) URLs can be specified.

- -host hostname:port, -path pathname

  if the host option is present then the OCSP request is sent to the host hostname on port port. path specifies the HTTP path name to use or "/" by default.

- -CAfile file, -CApath pathname

  file or pathname containing trusted CA certificates. These are used to verify the signature on the OCSP response.

- -verify_other file

  file containing additional certificates to search when attempting to locate the OCSP response signing certificate. Some responders omit the actual signer's certificate from the response: this option can be used to supply the necessary certificate in such cases.

- -trust_other

  the certificates specified by the -verify_certs option should be explicitly trusted and no additional checks will be performed on them. This is useful when the complete responder certificate chain is not available or trusting a root CA is not appropriate.

- -VAfile file

  file containing explicitly trusted responder certificates. Equivalent to the -verify_certs and -trust_other options.

- -noverify

  don't attempt to verify the OCSP response signature or the nonce values. This option will normally only be used for debugging since it disables all verification of the responders certificate.

- -no_intern

  ignore certificates contained in the OCSP response when searching for the signers certificate. With this option the signers certificate must be specified with either the -verify_certs or -VAfile options.

- -no_signature_verify

  don't check the signature on the OCSP response. Since this option tolerates invalid signatures on OCSP responses it will normally only be used for testing purposes.

- -no_cert_verify

  don't verify the OCSP response signers certificate at all. Since this option allows the OCSP response to be signed by any certificate it should only be used for testing purposes.

- -no_chain

  do not use certificates in the response as additional untrusted CA certificates.

- -no_cert_checks

  don't perform any additional checks on the OCSP response signers certificate. That is do not make any checks to see if the signers certificate is authorised to provide the necessary status information: as a result this option should only be used for testing purposes.

- -validity_period nsec, -status_age age

  these options specify the range of times, in seconds, which will be tolerated in an OCSP response. Each certificate status response includes a notBefore time and an optional notAfter time. The current time should fall between these two values, but the interval between the two times may be only a few seconds. In practice the OCSP responder and clients clocks may not be precisely synchronised and so such a check may fail. To avoid this the -validity_period option can be used to specify an acceptable error range in seconds, the default value is 5 minutes.

  If the notAfter time is omitted from a response then this means that new status information is immediately available. In this case the age of the notBefore field is checked to see it is not older than age seconds old. By default this additional check is not performed.

OCSP SERVER OPTIONS

- -index indexfile

  indexfile is a text index file in ca format containing certificate revocation information.

  If the index option is specified the ocsp utility is in responder mode, otherwise it is in client mode. The request(s) the responder processes can be either specified on the command line (using issuer and serial options), supplied in a file (using the respin option) or via external OCSP clients (if port or url is specified.)

  If the index option is present then the CA and rsigner options must also be present.

- -CA file

  CA certificate corresponding to the revocation information in indexfile.

- -rsigner file

  The certificate to sign OCSP responses with.

- -rother file

  Additional certificates to include in the OCSP response.

- -resp_no_certs

  Don't include any certificates in the OCSP response.

- -resp_key_id

  Identify the signer certificate using the key ID, default is to use the subject name.

- -rkey file

  The private key to sign OCSP responses with: if not present the file specified in the rsigner option is used.

- -port portnum

  Port to listen for OCSP requests on. The port may also be specified using the url option.

- -nrequest number

  The OCSP server will exit after receiving number requests, default unlimited.

- -nmin minutes, -ndays days

  Number of minutes or days when fresh revocation information is available: used in the nextUpdate field. If neither option is present then the nextUpdate field is omitted meaning fresh revocation information is immediately available.

OCSP Response Verification

OCSP Response follows the rules specified in RFC2560.

Initially the OCSP responder certificate is located and the signature on the OCSP request checked using the responder certificate's public key.

Then a normal certificate verify is performed on the OCSP responder certificate building up a certificate chain in the process. The locations of the trusted certificates used to build the chain can be specified by the CAfile and CApath options or they will be looked for in the standard OpenSSL certificates directory.

If the initial verify fails then the OCSP verify process halts with an error.

Otherwise the issuing CA certificate in the request is compared to the OCSP responder certificate: if there is a match then the OCSP verify succeeds.

Otherwise the OCSP responder certificate's CA is checked against the issuing CA certificate in the request. If there is a match and the OCSPSigning extended key usage is present in the OCSP responder certificate then the OCSP verify succeeds.

Otherwise the root CA of the OCSP responders CA is checked to see if it is trusted for OCSP signing. If it is the OCSP verify succeeds.

If none of these checks is successful then the OCSP verify fails.

What this effectively means if that if the OCSP responder certificate is authorised directly by the CA it is issuing revocation information about (and it is correctly configured) then verification will succeed.

If the OCSP responder is a "global responder" which can give details about multiple CAs and has its own separate certificate chain then its root CA can be trusted for OCSP signing. For example:

```
openssl x509 -in ocspCA.pem -addtrust OCSPSigning -out trustedCA.pem
```

Alternatively the responder certificate itself can be explicitly trusted with the -VAfile option.

NOTES

As noted, most of the verify options are for testing or debugging purposes. Normally only the -CApath, -CAfile and (if the responder is a 'global VA') -VAfile options need to be used.

The OCSP server is only useful for test and demonstration purposes: it is not really usable as a full OCSP responder. It contains only a very simple HTTP request handling and can only handle the POST form of OCSP queries. It also handles requests serially meaning it cannot respond to new requests until it has processed the current one. The text index file format of revocation is also inefficient for large quantities of revocation data.

It is possible to run the ocsp application in responder mode via a CGI script using the respin and respout options.

EXAMPLES

Create an OCSP request and write it to a file:

```
openssl ocsp -issuer issuer.pem -cert c1.pem -cert c2.pem -reqout req.der
```

Send a query to an OCSP responder with URL http://ocsp.myhost.com/ save the response to a file and print it out in text form

```
openssl ocsp -issuer issuer.pem -cert c1.pem -cert c2.pem \
    -url http://ocsp.myhost.com/ -resp_text -respout resp.der
```

Read in an OCSP response and print out text form:

```
openssl ocsp -respin resp.der -text
```

OCSP server on port 8888 using a standard ca configuration, and a separate responder certificate. All requests and responses are printed to a file.

```
openssl ocsp -index demoCA/index.txt -port 8888 -rsigner rcert.pem -CA demoCA/cacert.pem
-text -out log.txt
```

As above but exit after processing one request:

```
openssl ocsp -index demoCA/index.txt -port 8888 -rsigner rcert.pem -CA demoCA/cacert.pem
    -nrequest 1
```

Query status information using internally generated request:

```
openssl ocsp -index demoCA/index.txt -rsigner rcert.pem -CA demoCA/cacert.pem
    -issuer demoCA/cacert.pem -serial 1
```

Query status information using request read from a file, write response to a second file.

```
openssl ocsp -index demoCA/index.txt -rsigner rcert.pem -CA demoCA/cacert.pem
    -reqin req.der -respout resp.der
```

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`* `]`

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

- 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

COMMAND SUMMARY

The openssl program provides a rich variety of commands (*`command`* in the SYNOPSIS above), each of which often has a wealth of options and arguments (*`command_opts`* and *`command_args`* in the SYNOPSIS).

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. (no-*XXX* 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.
- `ocsp`

  Online Certificate Status Protocol utility.
- `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`

  This implements a generic SSL/TLS client which can establish a transparent connection to a remote server speaking SSL/TLS. It's 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's 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

PASS PHRASE ARGUMENTS

Several commands accept password arguments, typically using -passin and -passout 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 (like 'ps' under UNIX) this UNIX form should only be used where security is not important.

- env:var

  obtain the password from the environment variable var. Since the environment of other processes is visible on certain platforms (e.g. ps under certain UNIX OSes) 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. pathname need not refer to a regular file: it could for example refer to a device or named pipe.

- fd:number

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

- stdin

  read the password from standard input.

SEE ALSO

`asn1`*parse* (1), *ca* (1), *config* (5), *crl* (1), `crl2pkcs7(1)`, *dgst* (1), *dhparam* (1), *dsa* (1), *dsaparam* (1), *enc* (1), *gendsa* (1), *genrsa* (1), *nseq* (1), *openssl* (1), *passwd* (1), `pkcs12(1)`, `pkcs7(1)`, `pkcs8(1)`, *rand* (1), *req* (1), *rsa* (1), *rsautl* (1), *s_client* (1), *s_server* (1), *smime* (1), *spkac* (1), *verify* (1), *version* (1), `x509(1)`, *crypto* (3), *ssl* (3)

HISTORY

The *openssl* (1) document appeared in OpenSSL 0.9.2. The list-*XXX*-commands 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.

passwd

NAME

passwd – compute password hashes

Synopsis

`openssl passwd [-crypt] [-1] [-apr1] [-salt` *`string`*`] [-in` *`file`*`] [-stdin] [-noverify] [-quiet] [-table] {`*`password`*`}`

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 -in file, from stdin for option -stdin , or from the command line, or from the terminal otherwise. The UNIX standard algorithm crypt and the MD5-based BSD password algorithm 1 and its Apache variant apr1 are available.

OPTIONS

- -crypt

  Use the crypt algorithm (default).

- -1

  Use the MD5 based BSD password algorithm 1.

- -apr1

  Use the apr1 algorithm (Apache variant of the BSD algorithm).

- -salt *`string`*

  Use the specified salt. When reading a password from the terminal, this implies -noverify.

- -in *`file`*

  Read passwords from *`file`*.

- -stdin

  Read passwords from stdin.

- -noverify

  Don't verify when reading a password from the terminal.

- -quiet

  Don't output warnings when passwords given at the command line are truncated.

- -table

  In the output list, prepend the cleartext password and a TAB character to each password hash.

EXAMPLES

openssl passwd -crypt -salt xx password prints xxj31ZMTZzkVA.

openssl passwd -1 -salt xxxxxxxx password prints 1xxxxxxxx$UYCIxa628.9qXjpQCjM4a.

openssl passwd -apr1 -salt xxxxxxxx password prints $apr1$xxxxxxxx$dxHfLAsjHkDRmG83UXe8K0.

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 file(s)]
```

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.

COMMAND OPTIONS

There are a lot of options the meaning of some depends of 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 (see below).

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* (1).

- -passout arg

  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* (1).

- -noout

  this option 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 at all will be output.

- -nokeys

  no private keys will be output.

- -info

  output additional information about the PKCS#12 file structure, algorithms used and iteration counts.

- -des

  use DES to encrypt private keys before outputting.

- -des3

  use triple DES to encrypt private keys before outputting, this is the default.

- -idea

  use IDEA to encrypt private keys before outputting.

- -nodes

  don't encrypt the private keys at all.

- -nomacver

  don't attempt to verify the integrity MAC before reading the file.

- -twopass

  prompt 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

  This option specifies that a PKCS#12 file will be created rather than parsed.

- -out filename

  This specifies filename to write the PKCS#12 file to. 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 doesn't 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

  file to read private key from. If not present then a private key must be present in the input file.

- -name friendlyname

  This 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

  This 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 PHRASE ARGUMENTS section in *openssl* (1).

- -passin password

  pass phrase source to decrypt any input private keys with. For more information about the format of arg see the PASS PHRASE ARGUMENTS section in *openssl* (1).

- -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

  encrypt 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

  these options allow 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 file(s)

  a file or files containing random data used to seed the random number generator, or an EGD socket (see *RAND_egd* (3)). Multiple files can be specified separated by a OS-dependent character. The separator is ; for MS-Windows, , for OpenVMS, and : for all others.

NOTES

Although there are a large number of options most of them are very rarely used. For PKCS#12 file parsing only -in and -out need to be used for PKCS#12 file creation -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 outputting 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 -cacerts options to just output CA certificates.

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 can't handle triple DES encrypted private keys, then the option -keypbe PBE-SHA1-RC2-40 can be used to reduce the private key encryption to 40 bit RC2. A complete description of all algorithms is contained in the pkcs8 manual page.

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 info 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
```

Restrictions

Some would argue that the PKCS#12 standard is one big bug :-)

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 cannot 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
```

SEE ALSO

pkcs8(1)

pkcs7

NAME

pkcs7 – PKCS#7 utility

Synopsis

```
openssl pkcs7 [-inform PEM|DER] [-outform PEM|DER] [-in filename] [-out filename]
[-print_certs] [-text] [-noout] [-engine id]
```

DESCRIPTION

The pkcs7 command processes PKCS#7 files in DER or PEM format.

COMMAND OPTIONS

- -inform DER | PEM

  This specifies the input format. DER format is DER encoded PKCS#7 v1.5 structure.PEM (the default) is a base64 encoded version of the DER form with header and footer lines.

- -outform DER | PEM

  This specifies the output format, the options have the same meaning as the -inform option.

- -in filename

  This 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 certificates details in full rather than just subject and issuer names.

- -noout

  don't output the encoded version of the PKCS#7 structure (or certificates is -print_certs is set).

- -engine id

  specifying an engine (by it's unique id string) will cause req to attempt to obtain a functional reference to the specified engine, thus initialising it if needed. The engine will then be set as the default for all available algorithms.

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
```

NOTES

The PEM PKCS#7 format uses the header and footer lines:

```
-----BEGIN PKCS7-----
-----END PKCS7-----
```

For compatibility 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.

This PKCS#7 routines only understand PKCS#7 v 1.5 as specified in RFC2315 they cannot currently parse, for example, the new CMS as described in RFC2630.

SEE ALSO

`crl2pkcs7(1)`

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] [-engine id]
```

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.

COMMAND 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

  This 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

  This specifies the output format, the options have the same meaning as the -inform option.

- -in filename

  This specifies the input filename to read a key from or standard input if this option is not specified. If the key is encrypted a pass phrase will be prompted for.

- -passin arg

  the input file password source. For more information about the format of arg see the PASS PHRASE ARGUMENTS section in *openssl* (1).

- -out filename

  This specifies the output filename to write a key to or standard output by default. If any encryption options are set then a pass phrase will be prompted for. The output filename should not be the same as the input filename.

- -passout arg

  the output file password source. For more information about the format of arg see the PASS PHRASE ARGUMENTS section in *openssl* (1).

- -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 at all and should only be used when absolutely necessary. Certain software such as some versions of Java code signing software used unencrypted private keys.

- -nooct

  This option generates RSA private keys in a broken format that some software uses. Specifically the private key should be enclosed in a OCTET STRING but some software just includes the structure itself without the surrounding OCTET STRING.

- -embed

  This option 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

  This option 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

  This option 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 yet. If you are just using private keys with OpenSSL then this doesn't matter.

  The alg argument is the encryption algorithm to use, valid values include des, des3 and rc2. It is recommended that des3 is used.

- -v1 alg

  This option specifies a PKCS#5 v1.5 or PKCS#12 algorithm to use. A complete list of possible algorithms is included below.

- -engine id

  specifying an engine (by it's unique id string) will cause req to attempt to obtain a functional reference to the specified engine, thus initialising it if needed. The engine will then be set as the default for all available algorithms.

NOTES

The encrypted form of a PEM encode 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 that those encrypted using the traditional SSLeay compatible formats. So if additional security is considered 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.

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.

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
```

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 and 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.

Restrictions

There should be an option that prints out the encryption algorithm in use and other details such as the iteration count.

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 at present.

SEE ALSO

dsa (1), *rsa* (1), *genrsa* (1), *gendsa* (1)

rand

NAME

rand – generate pseudo-random bytes

Synopsis

`openssl rand [-out file] [-rand file(s)] [-base64] num`

DESCRIPTION

The rand command outputs *num* pseudo-random bytes after seeding the random number generator 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.

OPTIONS

- -out *file*

  Write to *file* instead of standard output.

- -rand *file(s)*

  Use specified file or files or EGD socket (see *RAND_egd* (3)) for seeding the random number generator. Multiple files can be specified separated by a OS-dependent character. The separator is ; for MS-Windows, , for OpenVMS, and :for all others.

- -base64

  Perform base64 encoding on the output.

SEE ALSO

RAND_bytes (3)

req

NAME

req – PKCS#10 certificate request and certificate generating utility.

Synopsis

```
openssl req [-inform PEM|DER] [-outform PEM|DER] [-in filename] [-passin arg] [-out
filename] [-passout arg] [-text] [-pubkey] [-noout] [-verify] [-modulus] [-new] [-rand
file(s)] [-newkey rsa:bits] [-newkey dsa:file] [-nodes] [-key filename] [-keyform PEM|DER]
[-keyout filename] [-[md5|sha1|md2|mdc2]] [-config filename] [-subj arg] [-x509] [-days n]
[-set_serial n] [-asn1-kludge] [-newhdr] [-extensions section] [-reqexts section] [-utf8]
[-nameopt] [-batch] [-verbose] [-engine id]
```

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.

COMMAND OPTIONS

- -inform DER | PEM

  This 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 PEM form is the default format: it consists of the DER format base64 encoded with additional header and footer lines.

- -outform DER | PEM

  This specifies the output format, the options have the same meaning as the -inform option.

- -in filename

  This 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

  the input file password source. For more information about the format of arg see the PASS PHRASE ARGUMENTS section in *openssl* (1).

- -out filename

  This specifies the output filename to write to or standard output by default.

- -passout arg

  the output file password source. For more information about the format of arg see the PASS PHRASE ARGUMENTS section in *openssl* (1).

- -text

  prints out the certificate request in text form.

- -pubkey

  outputs the public key.

- -noout

  this option prevents output of the encoded version of the request.

- -modulus

  this option prints out the value of the modulus of the public key contained in the request.

- -verify

  verifies the signature on the request.

- -new

  this option 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 file(s)

  a file or files containing random data used to seed the random number generator, or an EGD socket (see *RAND_egd* (3)). Multiple files can be specified separated by a OS-dependent character. The separator is ; for MS-Windows, , for OpenVMS, and : for all others.

- -newkey arg

  this option creates a new certificate request and a new private key. The argument takes one of two forms. rsa:nbits, where nbits is the number of bits, generates an RSA key nbits in size. dsa:filename generates a DSA key using the parameters in the file filename.

- -key filename

  This 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 argument. PEM is the default.

- -keyout filename

  this 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]

  this 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

  this allows an alternative configuration file to be specified, this overrides the compile time filename or any specified in the OPENSSL_CONF environment variable.

- -subj arg

  sets subject name for new request or supersedes the subject name when processing a request. The arg must be formatted as `/type0=value0/type1=value1/type2=...`, characters may be escaped by \ (backslash), no spaces are skipped.

- -x509

  this option 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. Unless specified using the set_serial option 0 will be used for the serial number.

- -days n

  when the -x509 option is being used this specifies the number of days to certify the certificate for. The default is 30 days.

- -set_serial n

  serial number to use when outputting a self signed certificate. This may be specified as a decimal value or a hex value if preceded by 0x. It is possible to use negative serial numbers but this is not recommended.

- -extensions section

- -reqexts section

  these options specify 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.

- -utf8

  this option causes field values to be interpreted as UTF8 strings, by default they are interpreted as ASCII. This means that the field values, whether prompted from a terminal or obtained from a configuration file, must be valid UTF8 strings.

- -nameopt option

  option which determines how the subject or issuer names are displayed. The option argument can be a single option or multiple options separated by commas. Alternatively the -nameopt switch may be used more than once to set multiple options. See the `x509(1)` manual page for details.

- -asn1-kludge

  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.

- -batch

  non-interactive mode.

- -verbose

  print extra details about the operations being performed.

- -engine id

  specifying an engine (by it's unique id string) will cause req to attempt to obtain a functional reference to the specified engine, thus initialising it if needed. The engine will then be set as the default for all available algorithms.

CONFIGURATION FILE FORMAT

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

  This 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

  This is 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

  This 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

  This 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

  This specifies a filename in which random number seed information is placed and read from, or an EGD socket (see *RAND_egd* (3)). 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

  This option 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

  This option 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 default which is also 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 just uses PrintableStrings and T61Strings: certain software has problems with BMPStrings and UTF8Strings: in particular Netscape.

- req_extensions

  this 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 switch.

- x509_extensions

  this specifies the configuration file section containing a list of extensions to add to certificate generated when the -x509 switch is used. It can be overridden by the -extensions command line switch.

- prompt

  if set to the value no this disables prompting of certificate fields and just takes values from the config file directly. It also changes the expected format of the distinguished_name and attributes sections.

- utf8

  if set to the value yes then field values to be interpreted as UTF8 strings, by default they are interpreted as ASCII. This means that the field values, whether prompted from a terminal or obtained from a configuration file, must be valid UTF8 strings.

- attributes

  this 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

  This 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 ATTRIBUTE SECTION FORMAT

There are two separate formats for the distinguished name and attribute sections. If the prompt option is set to no then these sections just consist of field names and values: for example,

```
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 just 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 of the form:

```
fieldName="prompt"
fieldName_default="default field value"
fieldName_min= 2
fieldName_max= 4
```

"fieldName" is the field name being used, for example 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 just 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 ever 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, stateOrProvinceName. Additionally emailAddress is include 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.

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:

```
1.2.3.4shortNameA longer Name
1.2.3.6otherNameOther 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
```

NOTES

The header and footer lines in the PEM format are normally:

```
-----BEGIN CERTIFICATE REQUEST-----
-----END CERTIFICATE REQUEST-----
```

some software (some versions of Netscape certificate server) instead needs:

```
-----BEGIN NEW CERTIFICATE REQUEST-----
-----END NEW CERTIFICATE REQUEST-----
```

which 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.

DIAGNOSTICS

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...

```
unable to find 'distinguished_name' in config
problems making Certificate Request
```

The first error message is the clue: it can't find the configuration file! Certain operations (like examining a certificate request) don't need a configuration file so its use isn't enforced. Generation of certificates or requests however does need a configuration file. This could be regarded as a bug.

Another puzzling 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 just 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 -asn1-kludge for more information.

ENVIRONMENT VARIABLES

The variable OPENSSL_CONF if defined allows an alternative configuration file location to be specified, it will be overridden by the -config command line switch if it is present. For compatibility reasons the SSLEAY_CONF environment variable serves the same purpose but its use is discouraged.

Restrictions

OpenSSL's handling of T61Strings (aka TeletexStrings) is broken: it effectively treats them as ISO-8859-1 (Latin 1), Netscape and MSIE have similar behaviour. This can cause problems if you need characters that aren't available in PrintableStrings and you don't want to or can't 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 just entered. Other things like extensions in certificate requests are statically defined in the configuration file. Some of these: like an email address in subjectAltName should be input by the user.

SEE ALSO

`x509(1)`, *ca* (1), *genrsa* (1), *gendsa* (1), *config* (5)

rsa

NAME

rsa – RSA key processing tool

Synopsis

```
openssl rsa [-inform PEM|NET|DER] [-outform PEM|NET|DER] [-in filename] [-passin arg] [-out
filename] [-passout arg] [-sgckey] [-des] [-des3] [-idea] [-text] [-noout] [-modulus]
[-check] [-pubin] [-pubout] [-engine id]
```

DESCRIPTION

The rsa command processes RSA keys. They can be converted between various forms and their components printed out.

Note: This command uses the traditional SSLeay compatible format for private key encryption: newer applications should use the more secure PKCS#8 format using the pkcs8 utility.

COMMAND OPTIONS

- -inform DER | NET | PEM

  This specifies the input format. The DER option uses an ASN1 DER encoded form compatible with the PKCS#1 RSAPrivateKey or SubjectPublicKeyInfo format. The PEM form is the default format: it consists of the DER format base64 encoded with additional header and footer lines. On input PKCS#8 format private keys are also accepted. The NET form is a format is described in the NOTES section.

- -outform DER | NET | PEM

  This specifies the output format, the options have the same meaning as the -inform option.

- -in filename

  This specifies the input filename to read a key from or standard input if this option is not specified. If the key is encrypted a pass phrase will be prompted for.

- -passin arg

  the input file password source. For more information about the format of arg see the PASS PHRASE ARGUMENTS section in *openssl* (1).

- -out filename

  This specifies the output filename to write a key to or standard output if this option is not specified. If any encryption options are set then a pass phrase will be prompted for. The output filename should not be the same as the input filename.

- -passout password

  the output file password source. For more information about the format of arg see the PASS PHRASE ARGUMENTS section in *openssl* (1).

- -sgckey

  use the modified NET algorithm used with some versions of Microsoft IIS and SGC keys.

- -des | -des3 | -idea

  These options encrypt 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 the key is written in plain text. This means that using the rsa utility to read in an encrypted key with no encryption option can be used to remove the pass phrase from a key, or by setting the encryption options it can be use to add or change the pass phrase. These options can only be used with PEM format output files.

- -text

  prints out the various public or private key components in plain text in addition to the encoded version.

- -noout

  this option prevents output of the encoded version of the key.

- -modulus

  this option prints out the value of the modulus of the key.

- -check

  this option checks the consistency of an RSA private key.

- -pubin

  by default a private key is read from the input file: with this option a public key is read instead.

- -pubout

  by default a private key is output: with this option a public key will be output instead. This option is automatically set if the input is a public key.

- -engine id

  specifying an engine (by it's unique id string) will cause req to attempt to obtain a functional reference to the specified engine, thus initialising it if needed. The engine will then be set as the default for all available algorithms.

NOTES

The PEM private key format uses the header and footer lines:

```
-----BEGIN RSA PRIVATE KEY-----
-----END RSA PRIVATE KEY-----
```

The PEM public key format uses the header and footer lines:

```
-----BEGIN PUBLIC KEY-----
-----END PUBLIC KEY-----
```

The NET form is a format compatible with older Netscape servers and Microsoft IIS .key files, this uses unsalted RC4 for its encryption. It is not very secure and so should only be used when necessary.

Some newer version of IIS have additional data in the exported .key files. To use these with the utility, view the file with a binary editor and look for the string "private-key", then trace back to the byte sequence 0x30, 0x82 (this is an ASN1 SEQUENCE). Copy all the data from this point onwards to another file and use that as the input to the rsa utility with the -inform NET option. If you get an error after entering the password try the -sgckey option.

EXAMPLES

To remove the pass phrase on an RSA private key:

```
openssl rsa -in key.pem -out keyout.pem
```

To encrypt a private key using triple DES:

```
openssl rsa -in key.pem -des3 -out keyout.pem
```

To convert a private key from PEM to DER format:

```
openssl rsa -in key.pem -outform DER -out keyout.der
```

To print out the components of a private key to standard output:

```
openssl rsa -in key.pem -text -noout
```

To just output the public part of a private key:

```
openssl rsa -in key.pem -pubout -out pubkey.pem
```

Restrictions

The command line password arguments don't currently work with NET format.

There should be an option that automatically handles .key files, without having to manually edit them.

SEE ALSO

`pkcs8(1)`, *dsa* (1), *genrsa* (1), *gendsa* (1)

rsautl

NAME

rsautl – RSA utility

Synopsis

```
openssl rsautl [-in file] [-out file] [-inkey file] [-pubin] [-certin] [-sign] [-verify]
[-encrypt] [-decrypt] [-pkcs] [-ssl] [-raw] [-hexdump] [-asn1parse]
```

DESCRIPTION

The rsautl command can be used to sign, verify, encrypt and decrypt data using the RSA algorithm.

COMMAND OPTIONS

- -in filename

  This 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

  the 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

  sign the input data and output the signed result. This requires and RSA private key.
- -verify

  verify the input data and output the recovered data.
- -encrypt

  encrypt the input data using an RSA public key.
- -decrypt

  decrypt 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 dump the output data.

- -asn1parse

  asn1parse the output data, this is useful when combined with the -verify option.

NOTES

rsautl because it uses the RSA algorithm directly 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 analyse the signature of certificates using this utility in conjunction with asn1parse. Consider the self signed example in certs/pca-cert.pem . Running asn1parse as follows yields:

```
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 with:

```
openssl asn1parse -in pca-cert.pem -out sig -noout -strparse 614
```

The certificate public key can be extracted with:

openssl x509 -in test/testx509.pem -pubout -noout >pubkey.pem

The signature can be analysed 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. It can be seen that the digest used was md5. The actual part of the certificate that was signed can be extracted with:

```
openssl asn1parse -in pca-cert.pem -out tbs -noout -strparse 4
```

and its digest computed with:

```
openssl md5 -c tbs
MD5(tbs)= f3:46:9e:aa:1a:4a:73:c9:37:ea:93:00:48:25:08:b5
```

which it can be seen agrees with the recovered value above.

SEE ALSO

dgst (1), *rsa* (1), *genrsa* (1)

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] [-debug] [-msg]
[-nbio_test] [-state] [-nbio] [-crlf] [-ign_eof] [-quiet] [-ssl2] [-ssl3] [-tls1]
[-no_ssl2] [-no_ssl3] [-no_tls1] [-bugs] [-cipher cipherlist] [-starttls protocol]
[-engine id] [-rand file(s)]
```

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.

OPTIONS

- -connect host:port

  This 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 1 second between each read and write call.

- -showcerts

  display the whole server certificate chain: normally only the server certificate itself is displayed.

- -prexit

  print 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. Note: 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

  print extensive debugging information including a hex dump of all traffic.

- -msg

  show all protocol messages with hex dump.

- -nbio_test

  tests non-blocking I/O

- -nbio

  turns on non-blocking I/O

- -crlf

  this option translated a line feed from the terminal into CR+LF as required by some servers.

- -ign_eof

  inhibit shutting down the connection when end of file is reached in the input.

- -quiet

  inhibit printing of session and certificate information. This implicitly turns on -ign_eof as well.

- -ssl2, -ssl3, -tls1, -no_ssl2, -no_ssl3, -no_tlsl

  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 bug in SSL and TLS implementations. Adding this option enables various workarounds.

- -cipher cipherlist

  this 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.

- -starttls protocol

  send the protocol-specific message(s) to switch to TLS for communication. protocol is a keyword for the intended protocol. Currently, the only supported keywords are "smtp" and "pop3".

- -engine id

  specifying an engine (by it's unique id string) will cause s_client to attempt to obtain a functional reference to the specified engine, thus initialising it if needed. The engine will then be set as the default for all available algorithms.

- -rand file(s)

  a file or files containing random data used to seed the random number generator, or an EGD socket (see *RAND_egd* (3)). Multiple files can be specified separated by a OS-dependent character. The separator is ; for MS-Windows, , for OpenVMS, and : for all others.

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, and if the line begins with a Q or if end of file is reached, the connection will be closed down.

NOTES

s_client can be used to debug SSL servers. To connect to an SSL HTTP server the command:

```
openssl s_client -connect servername:443
```

would typically be used (https uses port 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 like no client certificate then the -bugs, -ssl2, -ssl3, -tls1, -no_ssl2, -no_ssl3, -no_tls1 can be tried in case it is a buggy server. In particular you should play with 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 command 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. Therefor 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 rather 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 really exit if the server verification fails.

The -prexit option is a bit of a hack. We should really report information whenever a session is renegotiated.

SEE ALSO

sess_id (1), *s_server* (1), *ciphers* (1)

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] [-msg] [-state] [-CApath directory] [-CAfile filename]
[-nocert] [-cipher cipherlist] [-quiet] [-no_tmp_rsa] [-ssl2] [-ssl3] [-tls1] [-no_ssl2]
[-no_ssl3] [-no_tls1] [-no_dhe] [-bugs] [-hack] [-www] [-WWW] [-HTTP] [-engine id]
[-id_prefix arg] [-rand file(s)]
```

DESCRIPTION

The s_server command implements a generic SSL/TLS server which listens for connections on a given port using SSL/TLS.

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 servers 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, -dkey keyname

  specify 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). As noted above 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

  print extensive debugging information including a hex dump of all traffic.

- -msg

  show all protocol messages with hex dump.

- -nbio_test

  tests non blocking I/O

- -nbio

  turns on non blocking I/O

- -crlf

  this option translated a line feed from the terminal into CR+LF.

- -quiet

  inhibit printing of session and certificate information.

- -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.

- -bugs

  there are several known bug in SSL and TLS implementations. Adding this option enables various workarounds.

- -hack

  this option enables a further workaround for some some early Netscape SSL code (?).

- -cipher cipherlist

  this 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 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.

- -HTTP

  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. The files loaded are assumed to contain a complete and correct HTTP response (lines that are part of the HTTP response line and headers must end with CRLF).

- -engine id

  specifying an engine (by it's unique id string) will cause s_server to attempt to obtain a functional reference to the specified engine, thus initialising it if needed. The engine will then be set as the default for all available algorithms.

- -id_prefix arg

  generate SSL/TLS session IDs prefixed by arg . This is mostly useful for testing any SSL/TLS code (eg. proxies) that wish to deal with multiple servers, when each of which might be generating a unique range of session IDs (eg. with a certain prefix).

- -rand file(s)

  a file or files containing random data used to seed the random number generator, or an EGD socket (see *RAND_egd* (3)). Multiple files can be specified separated by a OS-dependent character. The separator is ; for MS-Windows, , for OpenVMS, and : for all others.

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 listed below.

- q

  end the current SSL connection but still accept new connections.

- Q

  end the current SSL connection and exit.

- r

  renegotiate the SSL session.

- R

  renegotiate the SSL session and request a client certificate.

- P

  send some plain text down the underlying TCP connection: this should cause the client to disconnect due to a protocol violation.

- S

  print out some session cache status information.

NOTES

s_server can be used to debug SSL clients. To accept connections from a web browser the command:

```
openssl s_server -accept 443 -www
```

can be used for example.

Most web browsers (in particular Netscape and MSIE) only support RSA cipher suites, so they cannot connect to servers which don't 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 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 just 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

sess_id (1), *s_client* (1), *ciphers* (1)

s_time

NAME

s_time – SSL/TLS performance timing program

Synopsis

`openssl s_time [-connect host:port] [-www page] [-cert filename] [-key filename] [-CApath directory] [-CAfile filename] [-reuse] [-new] [-verify depth] [-nbio] [-time seconds] [-ssl2] [-ssl3] [-bugs] [-cipher cipherlist]`

DESCRIPTION

The *s_client* command implements a generic SSL/TLS client which connects to a remote host using SSL/TLS. It can request a page from the server and includes the time to transfer the payload data in its timing measurements. It measures the number of connections within a given timeframe, the amount of data transferred (if any), and calculates the average time spent for one connection.

OPTIONS

- *-connect host:port*

  This specifies the host and optional port to connect to.

- *-www page*

  This specifies the page to GET from the server. A value of '/' gets the index.htm[l] page. If this parameter is not specified, then *s_time* will only perform the handshake to establish SSL connections but not transfer any payload data.

- *-cert certname*

  The certificate to use, if one is requested by the server. The default is not to use a certificate. The file is in PEM format.

- *-key keyfile*

  The private key to use. If not specified then the certificate file will be used. The file is in PEM format.

- *-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.

- *-new*

  performs the timing test using a new session ID for each connection. If neither *-new* nor *-reuse* are specified, they are both on by default and executed in sequence.

- *-reuse*

  performs the timing test using the same session ID; this can be used as a test that session caching is working. If neither *-new* nor *-reuse* are specified, they are both on by default and executed in sequence.

- *-nbio*

  turns on non-blocking I/O.

- *-ssl2*, *-ssl3*

  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. The timing program is not as rich in options to turn protocols on and off as the *s_client* (1) program and may not connect to all servers.

  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 *-ssl3* option; others will only support SSL v2 and may need the *-ssl2* option.

- *-bugs*

  there are several known bug in SSL and TLS implementations. Adding this option enables various workarounds.

- *-cipher cipherlist*

  this 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* (1) command for more information.

- *-time length*

  specifies how long (in seconds) *s_time* should establish connections and optionally transfer payload data from a server. Server and client performance and the link speed determine how many connections *s_time* can establish.

NOTES

s_client can be used to measure the performance of an SSL connection. To connect to an SSL HTTP server and get the default page the command

```
openssl s_time -connect servername:443 -www / -CApath yourdir -CAfile yourfile.pem -cipher
commoncipher [-ssl3]
```

would typically be used (https uses port 443). 'commoncipher' is a cipher to which both client and server can agree, see the *ciphers* (1) command for details.

If the handshake fails then there are several possible causes, if it is nothing obvious like no client certificate then the *-bugs*, *-ssl2*, *-ssl3* options can be tried in case it is a buggy server. In particular you should play with 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* (1) 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 of *s_client* (1) 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. Therefor merely including a client certificate on the command line is no guarantee that the certificate works.

Restrictions

Because this program does not have all the options of the *s_client* (1) program to turn protocols on and off, you may not be able to measure the performance of all protocols with all servers.

The *-verify* option should really exit if the server verification fails.

SEE ALSO

s_client (1), *s_server* (1), *ciphers* (1)

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] [-context ID]
```

DESCRIPTION

The sess_id process the encoded version of the SSL session structure and optionally prints out SSL session details (for example 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.

- -inform DER | PEM

  This 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

  This specifies the output format, the options have the same meaning as the -inform option.

- -in filename

  This specifies the input filename to read session information from or standard input by default.

- -out filename

  This 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

  this option prevents output of the encoded version of the session.

- -context ID

  this option can set the session id so the output session information uses the supplied ID. The ID can be any string of characters. This option wont normally be used.

OUTPUT

Typical output:

```
SSL-Session:
    Protocol  : TLSv1
    Cipher    : 0016
    Session-ID: 871E62626C554CE95488823752CBD5F3673A3EF3DCE9C67BD916C809914B40ED
```

```
    Session-ID-ctx: 01000000
    Master-Key:
A7CEFC571974BE02CAC305269DC59F76EA9F0B180CB6642697A68251F2D2BB57E51DBBB4C7885573192AE9AEE220F
ACD
    Key-Arg   : None
    Start Time: 948459261
    Timeout   : 300 (sec)
    Verify return code 0 (ok)
```

Theses are described below in more detail.

- Protocol

  this is 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

  this is the SSL session master key.

- Key-Arg

  the key argument, this is only used in SSL v2.

- Start Time

  this is the session start time represented as an integer in standard UNIX format.

- Timeout

  the timeout in seconds.

- Verify return code

  this is the return code when an SSL client certificate is verified.

NOTES

The PEM encoded session format uses the 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.

SEE ALSO

ciphers (1), *s_server* (1)

smime

NAME

smime – S/MIME utility

Synopsis

```
openssl smime [-encrypt] [-decrypt] [-sign] [-verify] [-pk7out] [-des] [-des3] [-rc2-40]
[-rc2-64] [-rc2-128] [-in file] [-certfile file] [-signer file] [-recip file] [-inform
SMIME|PEM|DER] [-passin arg] [-inkey file] [-out file] [-outform SMIME|PEM|DER] [-content
file] [-to addr] [-from ad] [-subject s] [-text] [-rand file(s)] [cert.pem]...
```

DESCRIPTION

The smime command handles S/MIME mail. It can encrypt, decrypt, sign and verify S/MIME messages.

COMMAND OPTIONS

There are five operation options that set the type of operation to be performed. The meaning of the other options varies according to the operation type.

- -encrypt

  encrypt 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

  decrypt 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

  sign 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

  verify 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

  this specifies the input format for the PKCS#7 structure. The default is SMIME which reads an S/MIME format message.

  PEM and DER format change this to expect PEM and DER format PKCS#7 structures instead. This currently only affects the input format of the PKCS#7 structure, if no PKCS#7 structure is being input (for example with -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

  this specifies the output format for the PKCS#7 structure. The default is SMIME which write an S/MIME format message.

  PEM and DER format change this to write PEM and DER format PKCS#7 structures instead. This currently only affects the output format of the PKCS#7 structure, if no PKCS#7 structure is being output (for example with -verify or -decrypt) this option has no effect.

- -content filename

  This specifies a file containing the detached content, this is only useful with the -verify command. 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

  this option 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, only used with -verify.

- -CApath dir

  a directory containing trusted CA certificates, only used with -verify. 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.

- -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. Only used with -encrypt.

- -nointern

  when verifying a message normally 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

  do not verify the signers certificate of a signed message.

- -nochain

  do not do chain verification of signers certificates: that is don't use the certificates in the signed message as untrusted CAs.

- -nosigs

  don't try to verify the signatures on the message.

- -nocerts

  when signing a message the signer's certificate is normally included with this option it is excluded. This will reduce the size of the signed message but the verifier must have a copy of the signers certificate available locally (passed using the -certfile option for example).

- -noattr

  normally when a message is signed a set of attributes are included which include the signing time and supported symmetric algorithms. With this option they are not included.

- -binary

  normally 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. When this option is present no translation occurs. This is useful when handling binary data which may not be in MIME format.

- -nodetach

  when signing a message use opaque signing: 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 file

  allows additional certificates to be specified. When signing these will be included with the message. When verifying these will be searched for the signers certificates. The certificates should be in PEM format.

- -signer file

  the signers certificate when signing a message. If a message is being verified then the signers certificates will be written to this file if the verification was successful.

- -recip file

  the recipients certificate when decrypting a message. This certificate must match one of the recipients of the message or an error occurs.

- -inkey file

  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 -signer file.

- -passin arg

  the private key password source. For more information about the format of arg see the PASS PHRASE ARGUMENTS section in *openssl* (1).

- -rand file(s)

  a file or files containing random data used to seed the random number generator, or an EGD socket (see *RAND_egd* (3)). Multiple files can be specified separated by a OS-dependent character. The separator is ; for MS-Windows, , for OpenVMS, and : 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 the signers certificate's email address matches that specified in the From: address.

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 wont 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: see the examples section.

This version of the program only allows one signer per message but it will verify multiple signers on received messages. Some S/MIME clients choke 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. Strictly speaking these process PKCS#7 enveloped data: PKCS#7 encrypted data is used for other purposes.

EXIT CODES

- 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 and 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
```

Note: 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:

```
 -----BEGIN PKCS7-----
 -----END PKCS7-----
```

and using the command,

```
 openssl smime -verify -inform PEM -in signature.pem -content content.txt
```

alternatively you can base64 decode the signature and use

```
 openssl smime -verify -inform DER -in signature.der -content content.txt
```

Restrictions

The MIME parser isn't very clever: it seems to handle most messages that I've thrown at it but it may choke on others.

The code currently 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 database should be maintained of a certificates for each email address.

The code doesn't currently 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 current code can only handle S/MIME v2 messages, the more complex S/MIME v3 structures may cause parsing errors.

speed

NAME

speed – test 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]
```

DESCRIPTION

This command is used to test the performance of cryptographic algorithms.

OPTIONS

- -engine id

  specifying an engine (by it's unique id string) will cause speed to attempt to obtain a functional reference to the specified engine, thus initialising 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, speed tests those algorithms, otherwise all of the above are tested.

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] [-engine id]
```

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.

COMMAND OPTIONS

- -in filename

  This 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

  create an SPKAC file using the private key in keyfile. The -in, -nout, -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* (1).

- -challenge string

  specifies the challenge string if an SPKAC is being created.

- -spkac spkacname

  allows an alternative name form the variable containing the SPKAC. The default is "SPKAC". This option affects both generated and input SPKAC files.

- -spksect section

  allows an alternative name form the section containing the SPKAC. The default is the default section.

- -noout

  don't output the text version of the SPKAC (not used if an SPKAC is being created).

- -pubkey

  output the public key of an SPKAC (not used if an SPKAC is being created).

- -verify

  verifies the digital signature on the supplied SPKAC.

- -engine id

  specifying an engine (by it's unique id string) will cause req to attempt to obtain a functional reference to the specified engine, thus initialising it if needed. The engine will then be set as the default for all available algorithms.

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=
```

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".

SEE ALSO

ca (1)

verify

NAME

verify – Utility to verify certificates.

Synopsis

```
openssl verify [-CApath directory] [-CAfile file] [-purpose purpose] [-untrusted file]
[-help] [-issuer_checks] [-verbose] [-] [certificates]
```

DESCRIPTION

The verify command verifies certificate chains.

COMMAND 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 ("hash" is the hashed certificate subject name: see the -hash option of the x509 utility). Under UNIX the c_rehash script will automatically create symbolic links to a directory of certificates.

- -CAfile file

  A file of trusted certificates. The file should contain multiple certificates in PEM format concatenated together.

- -untrusted file

  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

  print extra information about the operations being performed.

- -issuer_checks

  print 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.

VERIFY OPERATION

The verify program uses the same functions as the internal SSL and S/MIME verification, therefore this description applies to these verify operations too.

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 whereas normally 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.

Firstly a certificate chain is built up starting from the supplied certificate and ending in the root CA. It is an error if the whole chain cannot be built up. The chain is built up by looking up the issuers 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' itself 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 issuers 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.

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.

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.

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.

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 certificate is considered valid. If any operation fails then the certificate is not valid.

DIAGNOSTICS

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 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.

An exhaustive 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: that is 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. 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. 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. 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.

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 recognised.

Previous versions of OpenSSL assume certificates with matching subject name are identical and mishandled them.

SEE ALSO

x509(1)

version

NAME

version – print OpenSSL version information

Synopsis

```
openssl version [-a] [-v] [-b] [-o] [-f] [-p]
```

DESCRIPTION

This command is used to print out version information about OpenSSL.

OPTIONS

- -a

  all information, this is the same as setting all the other flags.

- -v

  the current OpenSSL version.

- -b

  the date the current version of OpenSSL was built.

- -o

  option information: various options set when the library was built.

- -c

  compilation flags.

- -p

  platform setting.

- -d

  OPENSSLDIR setting.

NOTES

The output of openssl version -a would typically be used when sending in a bug report.

HISTORY

The -d option was added in OpenSSL 0.9.7.

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] [-set_serial n] [-signkey
filename] [-x509toreq] [-req] [-CA filename] [-CAkey filename] [-CAcreateserial]
[-CAserial filename] [-text] [-C] [-md2|-md5|-sha1|-mdc2] [-clrext] [-extfile filename]
[-extensions section] [-engine id]
```

DESCRIPTION

The x509 command is a multi purpose certificate utility. It can be used to display certificate information, convert certificates to various forms, sign certificate requests like a "mini CA" or edit certificate trust settings.

Since there are a large number of options they will split up into various sections.

OPTIONS

INPUT, OUTPUT AND GENERAL PURPOSE OPTIONS

- -inform DER|PEM|NET

  This 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 now obsolete.

- -outform DER|PEM|NET

  This specifies the output format, the options have the same meaning as the -inform option.

- -in filename

  This specifies the input filename to read a certificate from or standard input if this option is not specified.

- -out filename

  This 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.

- -engine id

  specifying an engine (by it's unique id string) will cause req to attempt to obtain a functional reference to the specified engine, thus initialising it if needed. The engine will then be set as the default for all available algorithms.

DISPLAY OPTIONS

Note: the -alias and -purpose options are also display options but are described in the TRUST SETTINGS section.

- -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.

- -certopt option

  customise the output format used with -text . The option argument can be a single option or multiple options separated by commas. The -certopt switch may be also be used more than once to set multiple options. See the TEXT OPTIONS section for more information.

- -noout

  this option prevents output of the encoded version of the request.

- -modulus

  this option 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. The option argument can be a single option or multiple options separated by commas. Alternatively the -nameopt switch may be used more than once to set multiple options. See the NAME OPTIONS section for more information.

- -email

  outputs the email address(es) if any.

- -startdate

  prints out the start date of the certificate, that is the notBefore date.

- -enddate

  prints out the expiry date of the certificate, that is the notAfter date.

- -dates

  prints out the start and expiry dates of a certificate.

- -fingerprint

  prints out the digest of the DER encoded version of the whole certificate (see digest options).

- -C

  this outputs the certificate in the form of a C source file.

TRUST SETTINGS

Please note these options are currently experimental and may well 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".

Normally 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 currently are only used with a root CA. They allow a finer control over the purposes the root CA can be used for. 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

  this 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 for example "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 currently 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

  this option performs tests on the certificate extensions and outputs the results. For a more complete description see the CERTIFICATE EXTENSIONS section.

SIGNING OPTIONS

The x509 utility can be used to sign certificates and requests: it can thus behave like a "mini CA".

- -signkey filename

  this option 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. 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

  delete any extensions from a certificate. This option is used when a certificate is being created from another certificate (for example 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 for. 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.

- -set_serial n

  specifies the serial number to use. This option can be used with either the -signkey or -CA options. If used in conjunction with the -CA option the serial number file (as specified by the -CAserial or -CAcreateserial options) is not used.

  The serial number can be decimal or hex (if preceded by 0x). Negative serial numbers can also be specified but their use is not recommended.

- -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: that is its issuer name is set to the subject name of the CA and it is digitally signed using the CAs 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

  with this option the CA serial number file is created if it does not exist: it will contain the serial number "02" and the certificate being signed will have the 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 switch determines how the subject and issuer names are displayed. If no nameopt switch 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. Only the first four will normally be used.

- compat

  use the old format. This is equivalent to specifying no name options at all.

- 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 specifying the 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 esc_ctrl , esc_msb, sep_multiline, spc_eq, lname and align.

- esc_2253

  escape the "special" characters required by RFC2253 in a field That is ,+"<>;. Additionally # is escaped at the beginning of a string and a space character at the beginning or end of a string.

- esc_ctrl

  escape control characters. That is 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

  escape 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

  convert all strings to UTF8 format first. This is required by RFC2253. If you are lucky enough to 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

  this option does not attempt to interpret multibyte characters in any way. That is 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

  show 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

  dump non character string types (for example OCTET STRING) if this option is not set then non character string types will be displayed as though each content octet represents a single character.

- dump_all

  dump all fields. This option when used with dump_der allows the DER encoding of the structure to be unambiguously determined.

- dump_unknown

  dump any field whose OID is not recognised by OpenSSL.

- sep_comma_plus, sep_comma_plus_space, sep_semi_plus_space, sep_multiline

  these options determine 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

  reverse 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

  these options alter how the field name is displayed. nofname does not display the field at all. sname uses the "short name" form (CN for commonName for example). lname uses the long form. oid represents the OID in numerical form and is useful for diagnostic purpose.

- align

  align field values for a more readable output. Only usable with sep_multiline.

- spc_eq

  places spaces round the = character which follows the field name.

TEXT OPTIONS

As well as customising the name output format, it is also possible to customise the actual fields printed using the certopt options when the text option is present. The default behaviour is to print all fields.

- compatible

  use the old format. This is equivalent to specifying no output options at all.

- no_header

  don't print header information: that is the lines saying "Certificate" and "Data".

- no_version

  don't print out the version number.

- no_serial

  don't print out the serial number.

- no_signame

  don't print out the signature algorithm used.

- no_validity

  don't print the validity, that is the notBefore and notAfter fields.

- no_subject

  don't print out the subject name.

- no_issuer

  don't print out the issuer name.

- no_pubkey

  don't print out the public key.

- no_sigdump

  don't give a hexadecimal dump of the certificate signature.

- no_aux

  don't print out certificate trust information.

- no_extensions

  don't print out any X509V3 extensions.

- ext_default

  retain default extension behaviour: attempt to print out unsupported certificate extensions.

- ext_error

  print an error message for unsupported certificate extensions.

- ext_parse

  ASN1 parse unsupported extensions.

- ext_dump

  hex dump unsupported extensions.

- ca_default

  the value used by the ca utility, equivalent to no_issuer, no_pubkey, no_header, no_version, no_sigdump and no_signame.

EXAMPLES

Note: in these examples the '\' means the example should be all 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,-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 clientAuth \
-setalias "Steve's Class 1 CA" -out trust.pem
```

NOTES

The PEM format uses the header and footer lines:

```
-----BEGIN CERTIFICATE-----
-----END CERTIFICATE-----
```

it will also handle files containing:

```
-----BEGIN X509 CERTIFICATE-----
-----END X509 CERTIFICATE-----
```

Trusted certificates have the 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 and 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.

CERTIFICATE EXTENSIONS

The -purpose option checks the certificate extensions and determines what the certificate can be used for. The actual checks done are rather complex and include various hacks and workarounds to handle broken certificates and software.

The same code is used when verifying untrusted certificates in chains so this section is useful if a chain is rejected by the verify code.

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 really 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 again 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.

A complete description of each test is given below. The comments about basicConstraints and keyUsage and V1 certificates above apply to all CA certificates.

- SSL Client

  The extended key usage extension must be absent or include the "web client authentication" OID. keyUsage 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. keyUsage 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.

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 behaviour described in the TRUST SETTINGS is currently being developed. It thus describes the intended behaviour rather than the current behaviour. It is hoped that it will represent reality in OpenSSL 0.9.5 and later.

SEE ALSO

req (1), *ca* (1), *genrsa* (1), *gendsa* (1), *verify* (1)

CRYPTO Application Programming Interface (API) Reference

This reference section includes the OpenSSL **Crypto** 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`

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_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()
```

ASN1_OBJECT_new

NAME

ASN1_OBJECT_new, ASN1_OBJECT_free – object allocation functions

Synopsis

```
ASN1_OBJECT *ASN1_OBJECT_new(void);
void ASN1_OBJECT_free(ASN1_OBJECT *a);
```

DESCRIPTION

The ASN1_OBJECT allocation routines, allocate and free an ASN1_OBJECT structure, which represents an ASN1 OBJECT IDENTIFIER.

ASN1_OBJECT_new() allocates and initializes a ASN1_OBJECT structure.

ASN1_OBJECT_free() frees up the *ASN1_OBJECT* structure *a*.

NOTES

Although ASN1_OBJECT_new() allocates a new ASN1_OBJECT structure it is almost never used in applications. The ASN1 object utility functions such as OBJ_nid2obj() are used instead.

RETURN VALUES

If the allocation fails, ASN1_OBJECT_new() returns *NULL* and sets an error code that can be obtained by *ERR_get_error* (3). Otherwise it returns a pointer to the newly allocated structure.

ASN1_OBJECT_free() returns no value.

SEE ALSO

ERR_get_error (3), d2i_ASN1_*OBJECT* (3)

HISTORY

ASN1_OBJECT_new() and ASN1_OBJECT_free() are available in all versions of SSLeay and OpenSSL.

ASN1_STRING_dup

NAME

ASN1_STRING_dup, ASN1_STRING_cmp, ASN1_STRING_set, ASN1_STRING_length,

ASN1_STRING_length_set, ASN1_STRING_type, ASN1_STRING_data – ASN1_STRING utility functions

Synopsis

```
int ASN1_STRING_length(ASN1_STRING *x);
unsigned char * ASN1_STRING_data(ASN1_STRING *x);
ASN1_STRING * ASN1_STRING_dup(ASN1_STRING *a);
int ASN1_STRING_cmp(ASN1_STRING *a, ASN1_STRING *b);
int ASN1_STRING_set(ASN1_STRING *str, const void *data, int len);
int ASN1_STRING_type(ASN1_STRING *x);
int ASN1_STRING_to_UTF8(unsigned char **out, ASN1_STRING *in);
```

DESCRIPTION

These functions allow an *ASN1_STRING* structure to be manipulated.

ASN1_STRING_length() returns the length of the content of *x*.

ASN1_STRING_data() returns an internal pointer to the data of *x*. Since this is an internal pointer it should *not* be freed or modified in any way.

ASN1_STRING_dup() returns a copy of the structure *a*.

ASN1_STRING_cmp() compares *a* and *b* returning 0 if the two are identical. The string types and content are compared.

ASN1_STRING_set() sets the data of string *str* to the buffer *data* or length *len*. The supplied data is copied. If *len* is -1 then the length is determined by strlen(data).

ASN1_STRING_type() returns the type of *x*, using standard constants such as *V_ASN1_OCTET_STRING*.

ASN1_STRING_to_UTF8() converts the string *in* to UTF8 format, the converted data is allocated in a buffer in **out*. The length of *out* is returned or a negative error code. The buffer **out* should be free using OPENSSL_free().

NOTES

Almost all ASN1 types in OpenSSL are represented as an *ASN1_STRING* structure. Other types such as *ASN1_OCTET_STRING* are simply typedefed to *ASN1_STRING* and the functions call the *ASN1_STRING* equivalents. *ASN1_STRING* is also used for some *CHOICE* types which consist entirely of primitive string types such as *DirectoryString* and *Time*.

These functions should *not* be used to examine or modify *ASN1_INTEGER* or *ASN1_ENUMERATED* types: the relevant *INTEGER* or *ENUMERATED* utility functions should be used instead.

In general it cannot be assumed that the data returned by ASN1_STRING_data() is null terminated or does not contain embedded nulls. The actual format of the data will depend on the actual string type itself: for example for and IA5String the data will be ASCII, for a BMPString two bytes per character in big endian format, UTF8String will be in UTF8 format.

Similar care should be take to ensure the data is in the correct format when calling ASN1_STRING_set().

RETURN VALUES

None.

SEE ALSO

ERR_get_error (3)

HISTORY

None.

ASN1_STRING_new

NAME

ASN1_STRING_new, ASN1_STRING_type_new, ASN1_STRING_free – ASN1_STRING allocation functions

Synopsis

```
ASN1_STRING * ASN1_STRING_new(void);
ASN1_STRING * ASN1_STRING_type_new(int type);
void ASN1_STRING_free(ASN1_STRING *a);
```

DESCRIPTION

ASN1_STRING_new() returns an allocated *ASN1_STRING* structure. Its type is undefined.

ASN1_STRING_type_new() returns an allocated *ASN1_STRING* structure of type *type*.

ASN1_STRING_free() frees up *a*.

NOTES

Other string types call the *ASN1_STRING* functions. For example ASN1_OCTET_STRING_new() calls ASN1_STRING_type(V_ASN1_OCTET_STRING).

RETURN VALUES

ASN1_STRING_new() and ASN1_STRING_type_new() return a valid ASN1_STRING structure or *NULL* if an error occurred.

ASN1_STRING_free() does not return a value.

SEE ALSO

ERR_get_error (3)

HISTORY

None.

ASN1_STRING_print_ex

NAME

ASN1_STRING_print_ex, ASN1_STRING_print_ex_fp – ASN1_STRING output routines. ,

Synopsis

```
#include <openssl/asn1.h>
int ASN1_STRING_print_ex(BIO *out, ASN1_STRING *str, unsigned long flags);
int ASN1_STRING_print_ex_fp(FILE *fp, ASN1_STRING *str, unsigned long flags);
int ASN1_STRING_print(BIO *out, ASN1_STRING *str);
```

DESCRIPTION

These functions output an *ASN1_STRING* structure. *ASN1_STRING* is used to represent all the ASN1 string types.

ASN1_STRING_print_ex() outputs *str* to *out*, the format is determined by the options *flags*. ASN1_STRING_print_ex_fp() is identical except it outputs to *fp* instead.

ASN1_STRING_print() prints *str* to *out* but using a different format to ASN1_STRING_print_ex(). It replaces unprintable characters (other than CR, LF) with '.'.

NOTES

ASN1_STRING_print() is a legacy function which should be avoided in new applications.

Although there are a large number of options frequently *ASN1_STRFLAGS_RFC2253* is suitable, or on UTF8 terminals *ASN1_STRFLAGS_RFC2253 & ~ASN1_STRFLAGS_ESC_MSB*.

The complete set of supported options for *flags* is listed below.

Various characters can be escaped. If *ASN1_STRFLGS_ESC_2253* is set the characters determined by RFC2253 are escaped. If *ASN1_STRFLGS_ESC_CTRL* is set control characters are escaped. If *ASN1_STRFLGS_ESC_MSB* is set characters with the MSB set are escaped: this option should *not* be used if the terminal correctly interprets UTF8 sequences.

Escaping takes several forms.

If the character being escaped is a 16 bit character then the form "\WXXXX" is used using exactly four characters for the hex representation. If it is 32 bits then "\UXXXXXXXX" is used using eight characters of its hex representation. These forms will only be used if UTF8 conversion is not set (see below).

Printable characters are normally escaped using the backslash '\' character. If *ASN1_STRFLGS_ESC_QUOTE* is set then the whole string is instead surrounded by double quote characters: this is arguably more readable than the backslash notation. Other characters use the "\XX" using exactly two characters of the hex representation.

If *ASN1_STRFLGS_UTF8_CONVERT* is set then characters are converted to UTF8 format first. If the terminal supports the display of UTF8 sequences then this option will correctly display multi byte characters.

If *ASN1_STRFLGS_IGNORE_TYPE* is set then the string type is not interpreted at all: everything is assumed to be one byte per character. This is primarily for debugging purposes and can result in confusing output in multi character strings.

If *ASN1_STRFLGS_SHOW_TYPE* is set then the string type itself is printed out before its value (for example "BMPSTRING"), this actually uses ASN1_tag2str().

The content of a string instead of being interpreted can be "dumped": this just outputs the value of the string using the form #XXXX using hex format for each octet.

If *ASN1_STRFLGS_DUMP_ALL* is set then any type is dumped.

Normally non character string types (such as OCTET STRING) are assumed to be one byte per character, if *ASN1_STRFLAGS_DUMP_UNKNOWN* is set then they will be dumped instead.

When a type is dumped normally just the content octets are printed, if *ASN1_STRFLGS_DUMP_DER* is set then the complete encoding is dumped instead (including tag and length octets).

ASN1_STRFLGS_RFC2253 includes all the flags required by RFC2253. It is equivalent to:
ASN1_STRFLGS_ESC_2253 | ASN1_STRFLGS_ESC_CTRL | ASN1_STRFLGS_ESC_MSB | ASN1_STRFLGS_UTF8_CONVERT | ASN1_STRFLGS_DUMP_UNKNOWN ASN1_STRFLGS_DUMP_DER

SEE ALSO

X509_*NAME_print_ex* (3), ASN1_tag2*str* (3)

HISTORY

None.

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 type 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 decrypt data if it is being read from.

BIOs can be joined together 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 (normally a source/sink BIO).

SEE ALSO

BIO_ctrl (3), `BIO_f_base64(3)`, *BIO_f_buffer* (3), *BIO_f_cipher* (3), *BIO_f_md* (3), *BIO_f_null* (3), *BIO_f_ssl* (3), *BIO_find_type* (3), *BIO_new* (3), *BIO_new_bio_pair* (3), *BIO_push* (3), *BIO_read* (3), *BIO_s_accept* (3), *BIO_s_bio* (3), *BIO_s_connect* (3), *BIO_s_fd* (3), *BIO_s_file* (3), *BIO_s_mem* (3), *BIO_s_null* (3), *BIO_s_socket* (3), *BIO_set_callback* (3), *BIO_should_retry* (3)

BIO_ctrl

NAME

BIO_ctrl, BIO_callback_ctrl, BIO_ptr_ctrl, BIO_int_ctrl, BIO_reset, BIO_seek, BIO_tell,

BIO_flush, BIO_eof, BIO_set_close, BIO_get_close, BIO_pending, BIO_wpending,

BIO_ctrl_pending, BIO_ctrl_wpending, BIO_get_info_callback, BIO_set_info_callback – 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

BIO_ctrl(), BIO_callback_ctrl(), BIO_ptr_ctrl() and BIO_int_ctrl() are BIO "control" operations taking arguments of various types. These functions are not normally 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 BIOs manual page as well as any special features of the standard calls.

BIO_reset() 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.

BIO_seek() resets a file related BIO's (that is file descriptor and FILE BIOs) file position pointer to *ofs* bytes from start of file.

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

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

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

BIO_set_close() 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.

BIO_get_close() returns the BIOs close flag.

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

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.

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

NOTES

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

The return values of BIO_pending() and BIO_wpending() may 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 structures internal buffers but it is not possible to determine this in a portably way. For other types of BIO they may 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 may still succeed if the chain ends in a FILE or file descriptor BIO.

Source/sink BIOs return an 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.

SEE ALSO

None.

BIO_f_base64

NAME

BIO_f_base64 – base64 BIO filter

Synopsis

```
#include <openssl/bio.h>
#include <openssl/evp.h>
BIO_METHOD *BIO_f_base64(void);
```

DESCRIPTION

BIO_f_base64() 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 BIO_gets() or BIO_puts().

BIO_flush() 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 flag BIO_FLAGS_BASE64_NO_NL 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.

RETURN VALUES

BIO_f_base64() 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;

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, 512) > 0)
BIO_write(bio_out, inbuf, inlen);

 BIO_free_all(bio);
```

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).

SEE ALSO

None.

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

BIO_f_buffer() 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.

BIO_get_buffer_num_lines() returns the number of lines currently buffered.

BIO_set_read_buffer_size(), BIO_set_write_buffer_size() and BIO_set_buffer_size() 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.

BIO_set_buffer_read_data() 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(). BIO_flush() may 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).

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

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

SEE ALSO

None.

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

BIO_f_cipher() 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 cipher routines EVP_CipherInit(), EVP_CipherUpdate() and EVP_CipherFinal().

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

BIO_flush() 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.

BIO_set_cipher() sets the cipher of BIO *b* to *cipher* using key *key* and IV *iv*. *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. BIO_get_cipher_status() is a BIO_ctrl() macro which can be called to determine whether the decryption operation was successful.

BIO_get_cipher_ctx() 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. If it is not then 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. BIO_get_cipher_status() should be called to determine if the decrypt was successful.

As always, 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() currently always returns 1.

EXAMPLES

None.

SEE ALSO

None.

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_ctx(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 digest routines EVP_DigestInit(), EVP_DigestUpdate() and EVP_DigestFinal().

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

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

BIO_reset() reinitialises a digest BIO.

BIO_set_md() 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.

BIO_get_md() places the a pointer to the digest BIOs digest method in *mdp*, it is a BIO_ctrl() macro.

BIO_get_md_ctx() returns the digest BIOs context into *mdcp*.

NOTES

The context returned by BIO_get_md_ctx() can be used in calls to EVP_DigestFinal() and also the signature routines EVP_SignFinal() and EVP_VerifyFinal().

The context returned by BIO_get_md_ctx() 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.

RETURN VALUES

BIO_f_md() returns the digest BIO method.

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

EXAMPLES

The following example creates a BIO chain containing an SHA1 and MD5 digest BIO and passes 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));
```

The next example digests data by reading through a chain instead:

```
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);
```

This next example retrieves the message digests from a BIO chain and outputs them. This could be used with the examples above.

```
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);
```

Restrictions

The lack of support for BIO_puts() and the non standard behaviour 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.

SEE ALSO

None.

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

As may be apparent a null filter BIO is not particularly useful.

RETURN VALUES

BIO_f_null() returns the null filter BIO method.

SEE ALSO

None.

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

BIO_f_ssl() returns the SSL BIO method. This is a filter BIO which is a wrapper round the OpenSSL SSL routines adding a BIO "flavour" to SSL I/O.

I/O performed on an SSL BIO communicates using the SSL protocol with the SSLs 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 BIO_push() it is automatically used as the SSL BIOs 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 will typically 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().

BIO_set_ssl() sets the internal SSL pointer of BIO *b* to *ssl* using the close flag *c*.

BIO_get_ssl() retrieves the SSL pointer of BIO *b*, it can then be manipulated using the standard SSL library functions.

BIO_set_ssl_mode() sets the SSL BIO mode to *client*. If *client* is 1 client mode is set. If *client* is 0 server mode is set.

BIO_set_ssl_renegotiate_bytes() 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. *num* must be at least 512 bytes.

BIO_set_ssl_renegotiate_timeout() sets the renegotiate timeout to *seconds*. When the renegotiate timeout elapses the session is automatically renegotiated.

BIO_get_num_renegotiates() returns the total number of session renegotiations due to I/O or timeout.

BIO_new_ssl() allocates an SSL BIO using SSL_CTX *ctx* and using client mode if *client* is non zero.

BIO_new_ssl_connect() creates a new BIO chain consisting of an SSL BIO (using *ctx*) followed by a connect BIO.

BIO_new_buffer_ssl_connect() creates a new BIO chain consisting of a buffering BIO, an SSL BIO (using *ctx*) and a connect BIO.

BIO_ssl_copy_session_id() 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.

BIO_ssl_shutdown() 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.

BIO_do_handshake() 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 call BIO_should_retry() 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 flag SSL_AUTO_RETRY can be set to disable this behaviour. 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 BIO_set_host() on the BIO returned by BIO_new_ssl_connect() 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

None.

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* (3).

```
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, &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);
```

Here is a simple server example. It 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, &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, "<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, "</pre>\r\n");

 /* Since there is a buffering BIO present we had better flush it */
 BIO_flush(sbio);

 BIO_free_all(sbio);
```

SEE ALSO

None.

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() 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 searched for. BIO_find_type() returns the next matching BIO or NULL if none is found.

Note: not all the *BIO_TYPE_** types above have corresponding BIO implementations.

BIO_next() returns the next BIO in a chain. It can be used to traverse all BIOs in a chain or used in conjunction with BIO_find_type() to find all BIOs of a certain type.

BIO_method_type() returns the type of a BIO.

RETURN VALUES

BIO_find_type() returns a matching BIO or NULL for no match.

BIO_next() returns the next BIO in a chain.

BIO_method_type() returns the type of the BIO *b*.

NOTES

BIO_next() was added to OpenSSL 0.9.6 to provide a 'clean' way to traverse a BIO chain or find multiple matches using BIO_find_type(). Previous versions had to use:

```
 next = bio->next_bio;
```

Restrictions

BIO_find_type() in OpenSSL 0.9.5a and earlier could not be safely passed a NULL pointer for the *b* argument.

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);
```

SEE ALSO

None.

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*.

BIO_set() sets the method of an already existing BIO.

BIO_free() frees up a single BIO, BIO_vfree() also frees up a single BIO but it does not return a value. Calling BIO_free() may 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.

BIO_free_all() frees up an entire BIO chain, it does not halt if an error occurs freeing up an individual BIO in the chain.

RETURN VALUES

BIO_new() returns a newly created BIO or NULL if the call fails.

BIO_set(), BIO_free() return 1 for success and 0 for failure.

BIO_free_all() and BIO_vfree() do not return values.

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 BIO_free() 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.

Normally 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 normally called BIO_s_*() and a filter BIO BIO_f_*();

EXAMPLE

Create a memory BIO:

```
BIO *mem = BIO_new(BIO_s_mem());
```

SEE ALSO

None.

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*.

BIO_pop() removes the BIO *b* from a chain and returns the next BIO in the chain, or NULL if there is no next BIO. The removed BIO then becomes a single BIO with no association with the original chain, it can thus be freed or attached to a different chain.

NOTES

The names of these functions are perhaps a little misleading. BIO_push() joins two BIO chains whereas BIO_pop() 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 BIO_push() and BIO_pop() on a BIO may have additional consequences (a control call is made to the affected BIOs) any effects will be noted in the descriptions of individual BIOs.

EXAMPLES

For these examples suppose *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 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*.

It should be noted that reading causes data to pass in the reverse direction, that is data is read from *f*, base64 *decoded* and digested by *md1* and *md2*. If the call:

```
BIO_pop(md2);
```

The call will return *b64* and the new chain will be *md1-b64-f* data can be written to *md1* as before.

RETURN VALUES

BIO_push() returns the end of the chain, *b*.

BIO_pop() returns the next BIO in the chain, or NULL if there is no next BIO.

SEE ALSO

None.

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

BIO_read() attempts to read *len* bytes from BIO *b* and places the data in *buf*.

BIO_gets() 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*. There are exceptions to this however, for example BIO_gets() on a digest BIO will calculate and return the digest and other BIOs may not support BIO_gets() at all.

BIO_write() attempts to write *len* bytes from *buf* to BIO *b*.

BIO_puts() attempts to write a null terminated string *buf* to BIO *b*

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.

NOTES

A 0 or -1 return is not necessarily an indication of an error. In particular when the source/sink is non-blocking or of a certain type it may merely be an indication that no data is currently 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 non blocking I/O so successive reads will request a retry instead of blocking.

See *BIO_should_retry* (3) 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 possible to work around this by adding a buffering BIO *BIO_f_buffer* (3) to the chain.

SEE ALSO

BIO_should_retry (3)

BIO_s_accept

NAME

BIO_s_accept, 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);
long BIO_set_accept_port(BIO *b, char *name);
char *BIO_get_accept_port(BIO *b);
BIO *BIO_new_accept(char *host_port);
long BIO_set_nbio_accept(BIO *b, int n);
long BIO_set_accept_bios(BIO *b, char *bio);
long BIO_set_bind_mode(BIO *b, long mode);
long BIO_get_bind_mode(BIO *b, long dummy);
#define BIO_BIND_NORMAL0
#define BIO_BIND_REUSEADDR_IF_UNUSED1
#define BIO_BIND_REUSEADDR2
int BIO_do_accept(BIO *b);
```

DESCRIPTION

BIO_s_accept() 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 (see below) is set up properly then the BIO waits for an incoming connection.

Accept BIOs support BIO_puts() but not BIO_gets().

If the close flag 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.

BIO_get_fd() and BIO_set_fd() can be called to retrieve or set the accept socket. See *BIO_s_fd* (3)

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, that is 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: that is 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 non blocking mode if *n* is 1.

BIO_set_accept_bios() 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.

BIO_set_bind_mode() and BIO_get_bind_mode() 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_UNUSED is set then and 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, or request a retry in non blocking mode.

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. That is 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 BIO_set_accept_bios() then they are placed between the socket and the accept BIO, that is 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 now 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 BIO_free().

If only a single connection will be processed it is possible to perform I/O using the accept BIO itself. This is often undesirable however because the accept BIO will still accept additional incoming connections. This can be resolved by using BIO_pop() (see above) and freeing up the accept BIO after the initial connection.

If the underlying accept socket is non-blocking and BIO_do_accept() is called to await an incoming connection it is possible for BIO_should_io_special() with the reason BIO_RR_ACCEPT. If this happens then it is an indication that an accept attempt would block: the application should take appropriate action to wait until the underlying socket has accepted a connection and retry the call.

BIO_set_accept_port(), BIO_get_accept_port(), BIO_set_nbio_accept(), BIO_set_accept_bios(), BIO_set_bind_mode(), BIO_get_bind_mode() and BIO_do_accept() are macros.

RETURN VALUES

None.

EXAMPLE

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 (0);
 }

 /* Wait for incoming connection */
 if(BIO_do_accept(abio) <= 0) {
fprintf(stderr, "Error accepting connection\n");
ERR_print_errors_fp(stderr);
exit (0);
 }
 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 (0);
 }
 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);
```

SEE ALSO

None.

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

BIO_s_bio() 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.

BIO_reset() clears any data in the write buffer.

BIO_make_bio_pair() joins two separate BIOs into a connected pair.

BIO_destroy_pair() destroys the association between two connected BIOs. Freeing up any half of the pair will automatically destroy the association.

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.

BIO_set_write_buf_size() 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.

BIO_get_write_buf_size() returns the size of the write buffer.

BIO_new_bio_pair() combines the calls to BIO_new(), BIO_make_bio_pair() and BIO_set_write_buf_size() to create a connected pair of BIOs *bio1*, *bio2* with write buffer sizes *writebuf1* and *writebuf2*. If either size is zero then the default size is used. BIO_new_bio_pair() does not check whether *bio1* or *bio2* do point to some other BIO, the values are overwritten, BIO_free() is not called.

BIO_get_write_guarantee() and BIO_ctrl_get_write_guarantee() return the maximum length of data that can be currently 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. Note that 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. That is 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!

RETURN VALUES

BIO_new_bio_pair() returns 1 on success, with the new BIOs available in *bio1* and *bio2*, or 0 on failure, with NULL pointers 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, 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(), to find out whether data is buffered in the BIO and must be transfered to the network. Use BIO_ctrl_get_read_request() to find out, how many bytes must be written into the buffer before the SSL_operation() can successfully be continued.

WARNING

As the data is buffered, SSL_operation() may return with a ERROR_SSL_WANT_READ condition, but there is still data in the write buffer. An application must not rely on the error value of SSL_operation() 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.

SEE ALSO

SSL_set_bio (3), *ssl* (3), *bio* (3), *BIO_should_retry* (3), *BIO_read* (3)

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);
BIO *BIO_new_connect(char *name);
long BIO_set_conn_hostname(BIO *b, char *name);
long BIO_set_conn_port(BIO *b, char *port);
long BIO_set_conn_ip(BIO *b, char *ip);
long BIO_set_conn_int_port(BIO *b, char *port);
char *BIO_get_conn_hostname(BIO *b);
char *BIO_get_conn_port(BIO *b);
char *BIO_get_conn_ip(BIO *b, dummy);
long BIO_get_conn_int_port(BIO *b, int port);
long BIO_set_nbio(BIO *b, long n);
int BIO_do_connect(BIO *b);
```

DESCRIPTION

BIO_s_connect() 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 (see below) 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.

BIO_get_fd() 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".

BIO_set_conn_port() sets the port to *port*. *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.

BIO_set_conn_ip() sets the IP address to *ip* using binary form, that is four bytes specifying the IP address in big-endian form.

BIO_set_conn_int_port() sets the port using *port*. *port* should be of type (int *).

BIO_get_conn_hostname() 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.

BIO_get_conn_port() returns the port as a string.

BIO_get_conn_ip() returns the IP address in binary form.

BIO_get_conn_int_port() returns the port as an int.

BIO_set_nbio() 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 non blocking I/O is set during the connect process.

BIO_new_connect() combines BIO_new() and BIO_set_conn_hostname() into a single call: that is it creates a new connect BIO with *name*.

BIO_do_connect() 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 call BIO_should_retry() should be used for non blocking connect BIOs to determine if the call should be retried.

NOTES

If blocking I/O is set then a non positive return value from any I/O call is caused by an error condition, although 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 may 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 ':' 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 non blocking 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.

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() and BIO_do_connect() are macros.

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.

EXAMPLE

This is 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);
```

SEE ALSO

None.

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

BIO_s_fd() returns the file descriptor BIO method. This is a wrapper round the platforms file descriptor routines such as read() and write().

BIO_read() and BIO_write() 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.

BIO_reset() attempts to change the file pointer to the start of file using lseek(fd, 0, 0).

BIO_seek() sets the file pointer to position *ofs* from start of file using lseek(fd, ofs, 0).

BIO_tell() returns the current file position by calling lseek(fd, 0, 1).

BIO_set_fd() sets the file descriptor of BIO *b* to *fd* and the close flag to *c*.

BIO_get_fd() 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 *).

BIO_new_fd() returns a file descriptor BIO using *fd* and *close_flag*.

NOTES

The behaviour of BIO_read() and BIO_write() depends on the behavior of the platforms read() and write() calls on the descriptor. If the underlying file descriptor is in a non blocking mode then the BIO will behave in the manner described in the *BIO_read* (3) and *BIO_should_retry* (3) manual 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() behaviour.

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.

EXAMPLE

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

BIO_seek (3), *BIO_tell* (3), *BIO_reset* (3), *BIO_read* (3), *BIO_write* (3), *BIO_puts* (3), *BIO_gets* (3), *BIO_printf* (3), *BIO_set_close* (3), *BIO_get_close* (3)

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

BIO_s_file() 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() set 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 *name* for 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 behaviour will be mirrored by the corresponding BIO.

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);
```

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 return 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.

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 non positive value if an error occurred.

SEE ALSO

BIO_seek (3), *BIO_tell* (3), *BIO_reset* (3), *BIO_flush* (3), *BIO_read* (3), *BIO_write* (3), *BIO_puts* (3), *BIO_gets* (3), *BIO_printf* (3), *BIO_set_close* (3), *BIO_get_close* (3)

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: that is 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 a lots of 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.

The copying operation should not occur after every small read of a large BIO to improve efficiency.

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);
```

SEE ALSO

None.

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.

SEE ALSO

None.

BIO_s_socket

NAME

BIO_s_socket, BIO_new_socket – socket BIO

Synopsis

```
#include <openssl/bio.h>
BIO_METHOD *BIO_s_socket(void);
long BIO_set_fd(BIO *b, int fd, long close_flag);
long BIO_get_fd(BIO *b, int *c);
BIO *BIO_new_socket(int sock, int close_flag);
```

DESCRIPTION

BIO_s_socket() returns the socket BIO method. This is a wrapper round 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 *close_flag*.

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.

BIO_set_fd() and BIO_get_fd() are macros.

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.

SEE ALSO

None.

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 if is interpreted as a BIO to send the information to, otherwise stderr is used.

callback() is the callback function itself. The meaning of each argument is described below.

The BIO the callback is attached to is passed in *b*.

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.

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 immediately returned to the application and the BIO operation will not be performed.

The callback should normally simply return *retvalue* when it has finished processing, unless if 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.

EXAMPLE

The BIO_debug_callback() function is a good example, its source is in crypto/bio/bio_cb.c

SEE ALSO

None.

BIO_should_retry

NAME

BIO_should_retry, BIO_should_read, BIO_should_write, BIO_should_io_special, BIO_retry_type, BIO_should_retry, 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);
intBIO_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.

BIO_should_retry() 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.

BIO_should_read() is true if the cause of the condition is that a BIO needs to read data.

BIO_should_write() is true if the cause of the condition is that a BIO needs to read data.

BIO_should_io_special() is true if some "special" condition, that is a reason other than reading or writing is the cause of the condition.

BIO_get_retry_reason() returns a mask of the cause of a retry condition consisting of the values *BIO_FLAGS_READ*, *BIO_FLAGS_WRITE*, *BIO_FLAGS_IO_SPECIAL* though current BIO types will only set one of these.

BIO_get_retry_BIO() 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.

BIO_get_retry_reason() 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 may retry a failed non blocking 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 behaviour of the platforms I/O functions. This is often not desirable: one solution is to use non blocking I/O and use a timeout on the select() (or equivalent) call.

Restrictions

The OpenSSL ASN1 functions cannot gracefully deal with non blocking I/O: that is 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.

SEE ALSO

None.

blowfish

NAME

blowfish, BF_set_key, BF_encrypt, BF_decrypt, BF_ecb_encrypt, BF_cbc_encrypt, BF_cfb64_encrypt, 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 was 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 a considered good for strong encryption. Blowfish can be used in the same modes as DES (see *des_modes* (7)). Blowfish is currently one of the faster block ciphers. It is quite a bit faster than DES, and much faster than IDEA or RC2.

Blowfish consists of a key setup phase and the actual encryption or decryption phase.

BF_set_key() sets up the *BF_KEY key* using the *len* bytes long key at *data*.

BF_ecb_encrypt() 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*. *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 mode functions BF_cbc_encrypt(), BF_cfb64_encrypt() and BF_ofb64_encrypt() 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. *ivec* may be initialized with anything, but the recipient needs to know what it was initialized with, or it won't be able to decrypt. Some programs and protocols simplify this, like SSH, where *ivec* is simply initialized to zero. BF_cbc_encrypt() operates on data that is a multiple of 8 bytes long, while BF_cfb64_encrypt() and BF_ofb64_encrypt() are used to encrypt an 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.

BF_cbc_encrypt() 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*. *enc* decides if encryption (BF_ENCRYPT) or decryption (BF_DECRYPT) shall be performed. *ivec* must point at an 8 byte long initialization vector.

BF_cfb64_encrypt() 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*. *enc* decides if encryption (*BF_ENCRYPT*) or decryption (*BF_DECRYPT*) shall be performed. *ivec* must point at an 8 byte long initialization vector. *num* must point at an integer which must be initially zero.

BF_ofb64_encrypt() is the OFB mode for Blowfish with 64 bit feedback. It uses the same parameters as BF_cfb64_encrypt(), which must be initialized the same way.

BF_encrypt() and BF_decrypt() are the lowest level functions for Blowfish encryption. They encrypt/decrypt the first 64 bits of the vector pointed by *data*, using the key *key*. These functions should not be used unless you implement 'modes' of Blowfish. The alternative is to use BF_ecb_encrypt(). If you still want to use these functions, you should be aware that they take each 32-bit chunk in host-byte order, which is little-endian on little-endian platforms and big-endian on big-endian ones.

RETURN VALUES

None of the functions presented here return any value.

NOTE

Applications should use the higher level functions *EVP_EncryptInit* (3) etc. instead of calling the blowfish functions directly.

SEE ALSO

des_modes (7)

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);
BIGNUM *BN_swap(BIGNUM *a, BIGNUM *b);
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, 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_sqr(BIGNUM *r, BIGNUM *a, BN_CTX *ctx);
int BN_div(BIGNUM *dv, BIGNUM *rem, const BIGNUM *a, const BIGNUM *d, BN_CTX *ctx);
int BN_mod(BIGNUM *rem, const BIGNUM *a, const BIGNUM *m, BN_CTX *ctx);
int BN_nnmod(BIGNUM *rem, const BIGNUM *a, const BIGNUM *m, BN_CTX *ctx);
int BN_mod_add(BIGNUM *ret, BIGNUM *a, BIGNUM *b, const BIGNUM *m, BN_CTX *ctx);
int BN_mod_sub(BIGNUM *ret, BIGNUM *a, BIGNUM *b, const BIGNUM *m, BN_CTX *ctx);
int BN_mod_mul(BIGNUM *ret, BIGNUM *a, BIGNUM *b, const BIGNUM *m, BN_CTX *ctx);
int BN_mod_sqr(BIGNUM *ret, BIGNUM *a, 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);
const 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);
int BN_pseudo_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* (3); *BN_add* (3) describes most of the arithmetic operations. Comparison is described in *BN_cmp* (3); *BN_zero* (3) describes certain assignments, *BN_rand* (3) the generation of random numbers, *BN_generate_prime* (3) deals with prime numbers and *BN_set_bit* (3) with bit operations. The conversion of *BIGNUM*s to external formats is described in BN_bn2*bin* (3).

SEE ALSO

bn_internal (3), *dh* (3), *err* (3), *rand* (3), *rsa* (3), *BN_new* (3), *BN_CTX_new* (3), *BN_copy* (3), *BN_swap* (3), *BN_num_bytes* (3), *BN_add* (3), *BN_add_word* (3), *BN_cmp* (3), *BN_zero* (3), *BN_rand* (3), *BN_generate_prime* (3), *BN_set_bit* (3), BN_bn2*bin* (3), *BN_mod_inverse* (3), *BN_mod_mul_reciprocal* (3), *BN_mod_mul_montgomery* (3)

BN_add

NAME

BN_add, BN_sub, BN_mul, BN_sqr, BN_div, BN_mod, BN_nnmod, BN_mod_add, BN_mod_sub, BN_mod_mul, BN_mod_sqr, 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_sqr(BIGNUM *r, BIGNUM *a, BN_CTX *ctx);
int BN_div(BIGNUM *dv, BIGNUM *rem, const BIGNUM *a, const BIGNUM *d, BN_CTX *ctx);
int BN_mod(BIGNUM *rem, const BIGNUM *a, const BIGNUM *m, BN_CTX *ctx);
int BN_nnmod(BIGNUM *r, const BIGNUM *a, const BIGNUM *m, BN_CTX *ctx);
int BN_mod_add(BIGNUM *r, BIGNUM *a, BIGNUM *b, const BIGNUM *m, BN_CTX *ctx);
int BN_mod_sub(BIGNUM *r, BIGNUM *a, BIGNUM *b, const BIGNUM *m, BN_CTX *ctx);
int BN_mod_mul(BIGNUM *r, BIGNUM *a, BIGNUM *b, const BIGNUM *m, BN_CTX *ctx);
int BN_mod_sqr(BIGNUM *r, BIGNUM *a, 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

BN_add() adds *a* and *b* and places the result in *r* (r=a+b). *r* may be the same *BIGNUM* as *a* or *b*.

BN_sub() subtracts *b* from *a* and places the result in *r* (r=a-b).

BN_mul() multiplies *a* and *b* and places the result in *r* (r=a*b). *r* may be the same *BIGNUM* as *a* or *b*. For multiplication by powers of 2, use *BN_lshift* (3).

BN_sqr() takes the square of *a* and places the result in *r* (r=a^2). *r* and *a* may be the same *BIGNUM*. This function is faster than BN_mul(r,a,a).

BN_div() 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. The result is rounded towards zero; thus if *a* is negative, the remainder will be zero or negative. For division by powers of 2, use *BN_rshift* (3).

BN_mod() corresponds to BN_div() with *dv* set to *NULL*.

BN_nnmod() reduces *a* modulo *m* and places the non-negative remainder in *r*.

BN_mod_add() adds *a* to *b* modulo *m* and places the non-negative result in *r*.

BN_mod_sub() subtracts *b* from *a* modulo *m* and places the non-negative result in *r*.

BN_mod_mul() multiplies *a* by *b* and finds the non-negative remainder respective to modulus *m* (r=(a*b) mod m). *r* may be the same *BIGNUM* as *a* or *b*. For more efficient algorithms for repeated computations using the same modulus, see *BN_mod_mul_montgomery* (3) and *BN_mod_mul_reciprocal* (3).

BN_mod_sqr() takes the square of *a* modulo *m* and places the result in *r*.

BN_exp() raises *a* to the *p*-th 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*-th 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*. *r* 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* (3).

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 *ERR_get_error* (3).

SEE ALSO

bn (3), *ERR_get_error* (3), *BN_CTX_new* (3), *BN_add_word* (3), *BN_set_bit* (3)

HISTORY

BN_add(), BN_sub(), BN_sqr(), BN_div(), 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. BN_exp() appeared in SSLeay 0.9.0. BN_nnmod(), BN_mod_add(), BN_mod_sub(), and BN_mod_sqr() were added in OpenSSL 0.9.7.

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.

BN_add_word() adds *w* to *a* (`a+=w`).

BN_sub_word() subtracts *w* from *a* (`a-=w`).

BN_mul_word() multiplies *a* and *w* (`a*=b`).

BN_div_word() divides *a* by *w* (`a/=w`) and returns the remainder.

BN_mod_word() returns the remainder of *a* divided by *w* (`a%m`).

For BN_div_word() and BN_mod_word(), *w* must not be 0.

RETURN VALUES

BN_add_word(), BN_sub_word() and BN_mul_word() return 1 for success, 0 on error. The error codes can be obtained by *ERR_get_error* (3).

BN_mod_word() and BN_div_word() return *a%w*.

SEE ALSO

bn (3), *ERR_get_error* (3), *BN_add* (3)

HISTORY

BN_add_word() and BN_mod_word() are available in all versions of SSLeay and OpenSSL. BN_div_word() was added in SSLeay 0.8, and BN_sub_word() and BN_mul_word() in SSLeay 0.9.0.

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

BN_bn2bin() converts the absolute value of *a* into big-endian form and stores it at *to*. *to* must point to BN_num_bytes(*a*) bytes of memory.

BN_bin2bn() 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.

BN_bn2hex() and BN_bn2dec() 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.

BN_print() and BN_print_fp() write the hexadecimal encoding of *a*, with a leading '-' for negative numbers, to the *BIO* or *FILE fp*.

BN_bn2mpi() and BN_mpi2bn() convert *BIGNUM*s from and to a format that consists of the number's length in bytes represented as a 4-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).

BN_bn2mpi() 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).

BN_mpi2bn() 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

BN_bn2bin() returns the length of the big-endian number placed at *to*. BN_bin2bn() returns the *BIGNUM*, NULL on error.

BN_bn2hex() and BN_bn2dec() return a null-terminated string, or NULL on error. BN_hex2bn() and BN_dec2bn() return the number's length in hexadecimal or decimal digits, and 0 on error.

BN_print_fp() and BN_print() return 1 on success, 0 on write errors.

BN_bn2mpi() returns the length of the representation. BN_mpi2bn() returns the *BIGNUM*, and NULL on error.

The error codes can be obtained by *ERR_get_error* (3).

SEE ALSO

bn (3), *ERR_get_error* (3), *BN_zero* (3), ASN1_*INTEGER_to_BN* (3), *BN_num_bytes* (3)

HISTORY

BN_bn2bin(), BN_bin2bn(), BN_print_fp() and BN_print() are available in all versions of SSLeay and OpenSSL.

BN_bn2hex(), BN_bn2dec(), BN_hex2bn(), BN_dec2bn(), BN_bn2mpi() and BN_mpi2bn() were added in SSLeay 0.9.0.

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

BN_cmp() compares the numbers *a* and *b*. BN_ucmp() compares their absolute values.

BN_is_zero(), BN_is_one() and BN_is_word() test if *a* equals 0, 1, or *w* respectively. BN_is_odd() tests if a is odd.

BN_is_zero(), BN_is_one(), BN_is_word() and BN_is_odd() are macros.

RETURN VALUES

BN_cmp() returns -1 if $a < b$, 0 if $a == b$ and 1 if $a > b$. BN_ucmp() is the same using the absolute values of *a* and *b*.

BN_is_zero(), BN_is_one() BN_is_word() and BN_is_odd() return 1 if the condition is true, 0 otherwise.

SEE ALSO

bn (3)

HISTORY

BN_cmp(), BN_ucmp(), BN_is_zero(), BN_is_one() and BN_is_word() are available in all versions of SSLeay and OpenSSL. BN_is_odd() was added in SSLeay 0.8.

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

BN_copy() copies *from* to *to*. BN_dup() creates a new *BIGNUM* containing the value *from*.

RETURN VALUES

BN_copy() returns *to* on success, NULL on error. BN_dup() returns the new *BIGNUM*, and NULL on error. The error codes can be obtained by *ERR_get_error* (3).

SEE ALSO

bn (3), *ERR_get_error* (3)

HISTORY

BN_copy() and BN_dup() are available in all versions of SSLeay and OpenSSL.

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.

BN_CTX_new() allocates and initializes a *BN_CTX* structure. BN_CTX_init() initializes an existing uninitialized *BN_CTX*.

BN_CTX_free() frees the components of the *BN_CTX*, and if it was created by BN_CTX_new(), also the structure itself. If *BN_CTX_start* (3) has been used on the *BN_CTX*, *BN_CTX_end* (3) 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* (3).

BN_CTX_init() and BN_CTX_free() have no return values.

SEE ALSO

bn (3), *ERR_get_error* (3), *BN_add* (3), *BN_CTX_start* (3)

HISTORY

BN_CTX_new() and BN_CTX_free() are available in all versions on SSLeay and OpenSSL. BN_CTX_init() was added in SSLeay 0.9.1b.

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 *BN_CTX_new* (3)) 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

BN_CTX_start() and BN_CTX_end() return no values.

BN_CTX_get() 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* (3).

SEE ALSO

BN_CTX_new (3)

HISTORY

BN_CTX_start(), BN_CTX_get() and BN_CTX_end() were added in OpenSSL 0.9.5.

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

BN_generate_prime() 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_checks*, 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

BN_generate_prime() returns the prime number on success, *NULL* otherwise.

BN_is_prime() 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 *ERR_get_error* (3).

SEE ALSO

bn (3), *ERR_get_error* (3), *rand* (3)

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. BN_is_prime_fasttest() was added in OpenSSL 0.9.5.

bn_mul_words

NAME

bn_mul_words, bn_mul_add_words, bn_sqr_words, bn_div_words, bn_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, int dna,int
dnb,BN_ULONG *tmp);
void bn_mul_part_recursive(BN_ULONG *r, BN_ULONG *a, BN_ULONG *b, int n, int tna,int tnb,
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 documents 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()ed 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`.

max is the size of the *d* array that has been allocated. *top* is the 'last' entry being used, so for a value of 4, bn.d[0]=4 and bn.top=1. *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 *BIGNUM*s is rather expensive when used in conjunction with repeated subroutine calls, the *BN_CTX* structure is used. This structure contains *BN_CTX_NUM BIGNUM*s, see *BN_CTX_start* (3).

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*, *dna*, *dnb*, *t*) operates on the word arrays *a* and *b* of length *n2+dna* and *n2+dnb* (*dna* and *dnb* are currently allowed to be 0 or negative) and the 2**n2* word arrays *r* and *t*. *n2* must be a power of 2. It computes *a***b* and places the result in *r*.

bn_mul_part_recursive(*r*, *a*, *b*, *n*, *tna*, *tnb*, *tmp*) operates on the word arrays *a* and *b* of length *n+tna* and *n+tnb* 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

bn_expand() ensures that *b* has enough space for a *bits* bit number. bn_wexpand() 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

bn_check_top() verifies that `((a)->top >= 0 && (a)->top E<= (a)->max)>. A violation will cause the program to abort.`

bn_print() prints *a* to stderr. bn_dump() prints *n* words at *d* (in reverse order, i.e. most significant word first) to stderr.

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

bn (3)

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

BN_mod_inverse() computes the inverse of *a* modulo *n* places the result in *r* (`(a*r)%n==1`). If *r* is NULL, a new *BIGNUM* is created.

ctx is a previously allocated *BN_CTX* used for temporary variables. *r* may be the same *BIGNUM* as *a* or *n*.

RETURN VALUES

BN_mod_inverse() returns the *BIGNUM* containing the inverse, and NULL on error. The error codes can be obtained by *ERR_get_error* (3).

SEE ALSO

bn (3), *ERR_get_error* (3), *BN_add* (3)

HISTORY

BN_mod_inverse() is available in all versions of SSLeay and OpenSSL.

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* (3) is called with suitable input, but they may be useful when several operations are to be performed using the same modulus.

BN_MONT_CTX_new() allocates and initializes a *BN_MONT_CTX* structure. BN_MONT_CTX_init() initializes an existing uninitialized *BN_MONT_CTX*.

BN_MONT_CTX_set() sets up the *mont* structure from the modulus *m* by precomputing its inverse and a value R.

BN_MONT_CTX_copy() copies the *BN_MONT_CTX* *from* to *to*.

BN_MONT_CTX_free() frees the components of the *BN_MONT_CTX*, and, if it was created by BN_MONT_CTX_new(), also the structure itself.

BN_mod_mul_montgomery() computes Mont(*a*,*b*):=*a***b**R^-1 and places the result in *r*.

BN_from_montgomery() performs the Montgomery reduction *r* = *a**R^-1.

BN_to_montgomery() computes Mont(*a*,R^2), i.e. *a**R. Note that *a* must be non-negative and smaller than the modulus.

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

BN_MONT_CTX_new() returns the newly allocated *BN_MONT_CTX*, and NULL on error.

BN_MONT_CTX_init() and BN_MONT_CTX_free() 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* (3).

WARNING

The inputs must be reduced modulo *m*, otherwise the result will be outside the expected range.

SEE ALSO

bn (3), *ERR_get_error* (3), *BN_add* (3), *BN_CTX_new* (3)

HISTORY

BN_MONT_CTX_new(), BN_MONT_CTX_free(), BN_MONT_CTX_set(), BN_mod_mul_montgomery(), BN_from_montgomery() and BN_to_montgomery() are available in all versions of SSLeay and OpenSSL.

BN_MONT_CTX_init() and BN_MONT_CTX_copy() were added in SSLeay 0.9.1b.

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

BN_mod_mul_reciprocal() can be used to perform an efficient *BN_mod_mul* (3) 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.

BN_RECP_CTX_new() allocates and initializes a *BN_RECP* structure. BN_RECP_CTX_init() initializes an existing uninitialized *BN_RECP*.

BN_RECP_CTX_free() frees the components of the *BN_RECP*, and, if it was created by BN_RECP_CTX_new(), also the structure itself.

BN_RECP_CTX_set() 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*.

BN_div_recp() 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

BN_RECP_CTX_new() returns the newly allocated *BN_RECP_CTX*, and NULL on error.

BN_RECP_CTX_init() and BN_RECP_CTX_free() 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* (3).

SEE ALSO

bn (3), *ERR_get_error* (3), *BN_add* (3), *BN_CTX_new* (3)

HISTORY

BN_RECP_CTX was added in SSLeay 0.9.0. Before that, the function BN_reciprocal() was used instead, and the BN_mod_mul_reciprocal() arguments were different.

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

BN_new() allocated and initializes a *BIGNUM* structure. BN_init() initializes an existing uninitialized *BIGNUM*.

BN_clear() 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.

BN_free() frees the components of the *BIGNUM*, and if it was created by BN_new(), also the structure itself. BN_clear_free() additionally overwrites the data before the memory is returned to the system.

RETURN VALUES

BN_new() returns a pointer to the *BIGNUM*. If the allocation fails, it returns *NULL* and sets an error code that can be obtained by *ERR_get_error* (3).

BN_init(), BN_clear(), BN_free() and BN_clear_free() have no return values.

SEE ALSO

bn (3), *ERR_get_error* (3)

HISTORY

BN_new(), BN_clear(), BN_free() and BN_clear_free() are available in all versions on SSLeay and OpenSSL. BN_init() was added in SSLeay 0.9.1b.

BN_num_bits

NAME

BN_num_bits, BN_num_bytes, 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.

SEE ALSO

bn (3)

HISTORY

BN_num_bytes(), BN_num_bits() and BN_num_bits_word() are available in all versions of SSLeay and OpenSSL.

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);
int BN_pseudo_rand_range(BIGNUM *rnd, BIGNUM *range);
```

DESCRIPTION

BN_rand() generates a cryptographically strong pseudo-random number of *bits* 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.

BN_pseudo_rand() 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.

BN_rand_range() generates a cryptographically strong pseudo-random number *rnd* in the range 0 <lt>= *rnd* < *range*. BN_pseudo_rand_range() does the same, but is based on BN_pseudo_rand(), and hence numbers generated by it are not necessarily unpredictable.

The PRNG must be seeded prior to calling BN_rand() or BN_rand_range().

RETURN VALUES

The functions return 1 on success, 0 on error. The error codes can be obtained by *ERR_get_error* (3).

SEE ALSO

bn (3), *ERR_get_error* (3), *rand* (3), *RAND_add* (3), *RAND_bytes* (3)

HISTORY

BN_rand() is available in all versions of SSLeay and OpenSSL. BN_pseudo_rand() was added in OpenSSL 0.9.5. The *top* == -1 case and the function BN_rand_range() were added in OpenSSL 0.9.6a. BN_pseudo_rand_range() was added in OpenSSL 0.9.6c.

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

BN_set_bit() sets bit *n* in *a* to 1 (`a|=(1<<n)`). The number is expanded if necessary.

BN_clear_bit() sets bit *n* in *a* to 0 (`a&=~(1<<n)`). An error occurs if *a* is shorter than *n* bits.

BN_is_bit_set() tests if bit *n* in *a* is set.

BN_mask_bits() truncates *a* to an *n* bit number (`a&=~((~0)>>n)`). An error occurs if *a* already is shorter than *n* bits.

BN_lshift() shifts *a* left by *n* bits and places the result in *r* (`r=a*2^n`). BN_lshift1() shifts *a* left by one and places the result in *r* (`r=2*a`).

BN_rshift() shifts *a* right by *n* bits and places the result in *r* (`r=a/2^n`). BN_rshift1() 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

BN_is_bit_set() 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 *ERR_get_error* (3).

SEE ALSO

bn (3), *BN_num_bytes* (3), *BN_add* (3)

HISTORY

BN_set_bit(), BN_clear_bit(), BN_is_bit_set(), BN_mask_bits(), BN_lshift(), BN_lshift1(), BN_rshift(), and BN_rshift1() are available in all versions of SSLeay and OpenSSL.

BN_swap

NAME

BN_swap – exchange BIGNUMs

Synopsis

```
#include <openssl/bn.h>
void BN_swap(BIGNUM *a, BIGNUM *b);
```

DESCRIPTION

BN_swap() exchanges the values of *a* and *b*.

bn (3)

HISTORY

BN_swap was added in OpenSSL 0.9.7.

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);
const BIGNUM *BN_value_one(void);
int BN_set_word(BIGNUM *a, unsigned long w);
unsigned long BN_get_word(BIGNUM *a);
```

DESCRIPTION

BN_zero(), BN_one() and BN_set_word() set *a* to the values 0, 1 and *w* respectively. BN_zero() and BN_one() are macros.

BN_value_one() returns a *BIGNUM* constant of value 1. This constant is useful for use in comparisons and assignment.

BN_get_word() returns *a*, if it can be represented as an unsigned long.

RETURN VALUES

BN_get_word() returns the value *a*, and 0xffffffffL if *a* cannot be represented as an unsigned long.

BN_zero(), BN_one() and BN_set_word() return 1 on success, 0 otherwise. BN_value_one() 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.

SEE ALSO

bn (3), BN_bn2*bin* (3)

HISTORY

BN_zero(), BN_one() and BN_set_word() are available in all versions of SSLeay and OpenSSL.
BN_value_one() and BN_get_word() were added in SSLeay 0.8.

BN_value_one() was changed to return a true const BIGNUM * in OpenSSL 0.9.7.

BUF_MEM_new

NAME

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;
```

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.

BUF_MEM_new() allocates a new buffer of zero size.

BUF_MEM_free() frees up an already existing buffer. The data is zeroed before freeing up in case the buffer contains sensitive data.

BUF_MEM_grow() changes the size of an already existing buffer to *len*. Any data already in the buffer is preserved if it increases in size.

BUF_strdup() 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() and so 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 BUF_strdup() should be freed up using the OPENSSL_free() function.

RETURN VALUES

BUF_MEM_new() returns the buffer or NULL on error.

BUF_MEM_free() has no return value.

BUF_MEM_grow() returns zero on error or the new size (i.e. *len*).

SEE ALSO

bio (3)

HISTORY

BUF_MEM_new(), BUF_MEM_free() and BUF_MEM_grow() are available in all versions of SSLeay and OpenSSL. BUF_strdup() was added in SSLeay 0.8.

CONF_modules_free

NAME

CONF_modules_free, CONF_modules_load, CONF_modules_unload – OpenSSL configuration cleanup functions

Synopsis

```
#include <openssl/conf.h>
void CONF_modules_free(void);
void CONF_modules_unload(int all);
void CONF_modules_finish(void);
```

DESCRIPTION

CONF_modules_free() closes down and frees up all memory allocated by all configuration modules.

CONF_modules_finish() calls each configuration modules *finish* handler to free up any configuration that module may have performed.

CONF_modules_unload() finishes and unloads configuration modules. If *all* is set to *0* only modules loaded from DSOs will be unloads. If *all* is *1* all modules, including builtin modules will be unloaded.

NOTES

Normally applications will only call CONF_modules_free() at application to tidy up any configuration performed.

RETURN VALUE

None of the functions return a value.

SEE ALSO

conf (5), *OPENSSL_config* (3), *CONF_modules_load_file* (3), *CONF_modules_load_file* (3)

HISTORY

CONF_modules_free(), CONF_modules_unload(), and CONF_modules_finish() first appeared in OpenSSL 0.9.7.

CONF_modules_load_file

NAME

CONF_modules_load_file, CONF_modules_load – OpenSSL configuration functions

Synopsis

```
#include <openssl/conf.h>
int CONF_modules_load_file(const char *filename, const char *appname, unsigned long
flags);
int CONF_modules_load(const CONF *cnf, const char *appname, unsigned long flags);
```

DESCRIPTION

The function CONF_modules_load_file() configures OpenSSL using file *filename* and application name *appname*. If *filename* is NULL the standard OpenSSL configuration file is used. If *appname* is NULL the standard OpenSSL application name *openssl_conf* is used. The behaviour can be cutomized using *flags*.

CONF_modules_load() is idential to CONF_modules_load_file() except it read configuration information from *cnf*.

NOTES

The following *flags* are currently recognized:

CONF_MFLAGS_IGNORE_ERRORS if set errors returned by individual configuration modules are ignored. If not set the first module error is considered fatal and no further modules are loads.

Normally any modules errors will add error information to the error queue. If *CONF_MFLAGS_SILENT* is set no error information is added.

If *CONF_MFLAGS_NO_DSO* is set configuration module loading from DSOs is disabled.

CONF_MFLAGS_IGNORE_MISSING_FILE if set will make CONF_load_modules_file() ignore missing configuration files. Normally a missing configuration file return an error.

RETURN VALUE

These functions return 1 for success and a zero or negative value for failure. If module errors are not ignored the return code will reflect the return value of the failing module (this will always be zero or negative).

SEE ALSO

conf (5), *OPENSSL_config* (3), *CONF_free* (3), *CONF_free* (3), *err* (3), *err* (3)

HISTORY

CONF_modules_load_file and CONF_modules_load first appeared in OpenSSL 0.9.7.

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.

OVERVIEW

libcrypto consists of a number of sub-libraries 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 (3), *cast* (3), *des* (3), *idea* (3), `rc2(3)`, `rc4(3)`, `rc5(3)`
- PUBLIC KEY CRYPTOGRAPHY AND KEY AGREEMENT

  dsa (3), *dh* (3), *rsa* (3)
- CERTIFICATES

  `x509(3)`, `x509v3(3)`
- AUTHENTICATION CODES, HASH FUNCTIONS

  hmac (3), `md2(3)`, `md4(3)`, `md5(3)`, `mdc2(3)`, *ripemd* (3), *sha* (3)
- AUXILIARY FUNCTIONS

  err (3), *threads* (3), *rand* (3), *OPENSSL_VERSION_NUMBER* (3)
- INPUT/OUTPUT, DATA ENCODING

  `asn1(3)`, *bio* (3), *evp* (3), *pem* (3), `pkcs7(3)`, `pkcs12(3)`
- INTERNAL FUNCTIONS

  bn (3), *buffer* (3), *lhash* (3), *objects* (3), *stack* (3), *txt_db* (3)

NOTES

Some of the newer functions follow a naming convention using the numbers *0* and *1*. For example the functions:

```
int X509_CRL_add0_revoked(X509_CRL *crl, X509_REVOKED *rev);
int X509_add1_trust_object(X509 *x, ASN1_OBJECT *obj);
```

The *0* version uses the supplied structure pointer directly in the parent and it will be freed up when the parent is freed. In the above example *crl* would be freed but *rev* would not.

The *1* function uses a copy of the supplied structure pointer (or in some cases increases its link count) in the parent and so both (*x* and *obj* above) should be freed up.

SEE ALSO

openssl (1), *ssl* (3)

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.

CRYPTO_set_ex_data() is used to set application specific data, the data is supplied in the *arg* parameter and its precise meaning is up to the application.

CRYPTO_get_ex_data() 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

CRYPTO_set_ex_data() returns 1 on success or 0 on failure.

CRYPTO_get_ex_data() 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.

On failure an error code can be obtained from *ERR_get_error* (3).

SEE ALSO

RSA_get_ex_new_index (3), *DSA_get_ex_new_index* (3), *DH_get_ex_new_index* (3)

HISTORY

CRYPTO_set_ex_data() and CRYPTO_get_ex_data() have been available since SSLeay 0.9.0.

d2i_ASN1_OBJECT

NAME

d2i_ASN1_OBJECT, i2d_ASN1_OBJECT – ASN1 OBJECT IDENTIFIER functions

Synopsis

```
#include <openssl/objects.h>
ASN1_OBJECT *d2i_ASN1_OBJECT(ASN1_OBJECT **a, unsigned char **pp, long length);
int i2d_ASN1_OBJECT(ASN1_OBJECT *a, unsigned char **pp);
```

DESCRIPTION

These functions decode and encode an ASN1 OBJECT IDENTIFIER.

Othewise these behave in a similar way to d2i_X509() and i2d_X509() described in the `d2i_X509(3)` manual page.

SEE ALSO

`d2i_X509(3)`

HISTORY

None.

d2i_DHparams

NAME

d2i_DHparams, i2d_DHparams – PKCS#3 DH parameter functions.

Synopsis

```
#include <openssl/dh.h>
DH *d2i_DHparams(DH **a, unsigned char **pp, long length);
int i2d_DHparams(DH *a, unsigned char **pp);
```

DESCRIPTION

These functions decode and encode PKCS#3 DH parameters using the DHparameter structure described in PKCS#3.

Othewise these behave in a similar way to d2i_X509() and i2d_X509() described in the `d2i_X509(3)` manual page.

SEE ALSO

`d2i_X509(3)`

HISTORY

None.

d2i_DSAPublicKey

NAME

d2i_DSAPublicKey, i2d_DSAPublicKey, d2i_DSAPrivateKey, i2d_DSAPrivateKey,

d2i_DSA_PUBKEY, i2d_DSA_PUBKEY, d2i_DSA_SIG, i2d_DSA_SIG – DSA key encoding and parsing functions.

Synopsis

```
#include <openssl/dsa.h>
DSA * d2i_DSAPublicKey(DSA **a, const unsigned char **pp, long length);
int i2d_DSAPublicKey(const DSA *a, unsigned char **pp);
DSA * d2i_DSA_PUBKEY(DSA **a, const unsigned char **pp, long length);
int i2d_DSA_PUBKEY(const DSA *a, unsigned char **pp);
DSA * d2i_DSAPrivateKey(DSA **a, const unsigned char **pp, long length);
int i2d_DSAPrivateKey(const DSA *a, unsigned char **pp);
DSA * d2i_DSAparams(DSA **a, const unsigned char **pp, long length);
int i2d_DSAparams(const DSA *a, unsigned char **pp);
DSA * d2i_DSA_SIG(DSA_SIG **a, const unsigned char **pp, long length);
int i2d_DSA_SIG(const DSA_SIG *a, unsigned char **pp);
```

DESCRIPTION

d2i_DSAPublicKey() and i2d_DSAPublicKey() decode and encode the DSA public key components structure.

d2i_DSA_PUKEY() and i2d_DSA_PUKEY() decode and encode an DSA public key using a SubjectPublicKeyInfo (certificate public key) structure.

d2i_DSAPrivateKey(), i2d_DSAPrivateKey() decode and encode the DSA private key components.

d2i_DSAparams(), i2d_DSAparams() decode and encode the DSA parameters using a *Dss-Parms* structure as defined in RFC2459.

d2i_DSA_SIG(), i2d_DSA_SIG() decode and encode a DSA signature using a *Dss-Sig-Value* structure as defined in RFC2459.

The usage of all of these functions is similar to the d2i_X509() and i2d_X509() described in the `d2i_X509(3)` manual page.

NOTES

The *DSA* structure passed to the private key encoding functions should have all the private key components present.

The data encoded by the private key functions is unencrypted and therefore offers no private key security.

The *DSA_PUBKEY* functions should be used in preference to the *DSAPublicKey* functions when encoding public keys because they use a standard format.

The *DSAPublicKey* functions use an non standard format the actual data encoded depends on the value of the *write_params* field of the *a* key parameter. If *write_params* is zero then only the *pub_key* field is encoded as an *INTEGER*. If *write_params* is 1 then a *SEQUENCE* consisting of the *p*, *q*, *g* and *pub_key* respectively fields are encoded.

The *DSAPrivateKey* functions also use a non standard structure consiting consisting of a SEQUENCE containing the *p*, *q*, *g* and *pub_key* and *priv_key* fields respectively.

SEE ALSO

`d2i_X509(3)`

HISTORY

None.

d2i_PKCS8PrivateKey_bio

NAME

d2i_PKCS8PrivateKey_bio, d2i_PKCS8PrivateKey_fp, i2d_PKCS8PrivateKey_bio,

i2d_PKCS8PrivateKey_fp, i2d_PKCS8PrivateKey_nid_bio, i2d_PKCS8PrivateKey_nid_fp – PKCS#8 format private key functions

Synopsis

```
#include <openssl/evp.h>
EVP_PKEY *d2i_PKCS8PrivateKey_bio(BIO *bp, EVP_PKEY **x, pem_password_cb *cb, void *u);
EVP_PKEY *d2i_PKCS8PrivateKey_fp(FILE *fp, EVP_PKEY **x, pem_password_cb *cb, void *u);
int i2d_PKCS8PrivateKey_bio(BIO *bp, EVP_PKEY *x, const EVP_CIPHER *enc, char *kstr, int
klen, pem_password_cb *cb, void *u);
int i2d_PKCS8PrivateKey_fp(FILE *fp, EVP_PKEY *x, const EVP_CIPHER *enc, char *kstr, int
klen, pem_password_cb *cb, void *u);
int i2d_PKCS8PrivateKey_nid_bio(BIO *bp, EVP_PKEY *x, int nid, char *kstr, int klen,
pem_password_cb *cb, void *u);
int i2d_PKCS8PrivateKey_nid_fp(FILE *fp, EVP_PKEY *x, int nid, char *kstr, int klen,
pem_password_cb *cb, void *u);
```

DESCRIPTION

The PKCS#8 functions encode and decode private keys in PKCS#8 format using both PKCS#5 v1.5 and PKCS#5 v2.0 password based encryption algorithms.

Other than the use of DER as opposed to PEM these functions are identical to the corresponding *PEM* function as described in the *pem* (3) manual page.

NOTES

Before using these functions *OpenSSL_add_all_algorithms* (3) should be called to initialize the internal algorithm lookup tables otherwise errors about unknown algorithms will occur if an attempt is made to decrypt a private key.

These functions are currently the only way to store encrypted private keys using DER format.

Currently all the functions use BIOs or FILE pointers, there are no functions which work directly on memory: this can be readily worked around by converting the buffers to memory BIOs, see *BIO_s_mem* (3) for details.

SEE ALSO

pem (3)

d2i_RSAPublicKey

NAME

d2i_RSAPublicKey, i2d_RSAPublicKey, d2i_RSAPrivateKey, i2d_RSAPrivateKey,

d2i_RSA_PUBKEY, i2d_RSA_PUBKEY, i2d_Netscape_RSA, d2i_Netscape_RSA – RSA public and private key encoding functions.

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_RSA_PUBKEY(RSA **a, unsigned char **pp, long length);
int i2d_RSA_PUBKEY(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

d2i_RSAPublicKey() and i2d_RSAPublicKey() decode and encode a PKCS#1 RSAPublicKey structure.

d2i_RSA_PUKEY() and i2d_RSA_PUKEY() decode and encode an RSA public key using a SubjectPublicKeyInfo (certificate public key) structure.

d2i_RSAPrivateKey(), i2d_RSAPrivateKey() decode and encode a PKCS#1 RSAPrivateKey structure.

d2i_Netscape_RSA(), i2d_Netscape_RSA() decode and encode an RSA private key in NET format.

The usage of all of these functions is similar to the d2i_X509() and i2d_X509() described in the `d2i_X509(3)` manual page.

NOTES

The *RSA* structure passed to the private key encoding functions should have all the PKCS#1 private key components present.

The data encoded by the private key functions is unencrypted and therefore offers no private key security.

The NET format functions are present to provide compatibility with certain very old software. This format has some severe security weaknesses and should be avoided if possible.

SEE ALSO

`d2i_X509(3)`

HISTORY

None.

d2i_X509

NAME

d2i_X509, i2d_X509, d2i_X509_bio, d2i_X509_fp, i2d_X509_bio, i2d_X509_fp – X509 encode and decode functions

Synopsis

```
#include <openssl/x509.h>
X509 *d2i_X509(X509 **px, unsigned char **in, int len);
int i2d_X509(X509 *x, unsigned char **out);
X509 *d2i_X509_bio(BIO *bp, X509 **x);
X509 *d2i_X509_fp(FILE *fp, X509 **x);
int i2d_X509_bio(X509 *x, BIO *bp);
int i2d_X509_fp(X509 *x, FILE *fp);
```

DESCRIPTION

The X509 encode and decode routines encode and parse an *X509* structure, which represents an X509 certificate.

d2i_X509() attempts to decode *len* bytes at **out*. If successful a pointer to the *X509* structure is returned. If an error occurred then *NULL* is returned. If *px* is not *NULL* then the returned structure is written to **px*. If **px* is not *NULL* then it is assumed that **px* contains a valid *X509* structure and an attempt is made to reuse it. If the call is successful **out* is incremented to the byte following the parsed data.

i2d_X509() encodes the structure pointed to by *x* into DER format. If *out* is not *NULL* is writes the DER encoded data to the buffer at **out*, and increments it to point after the data just written. If the return value is negative an error occurred, otherwise it returns the length of the encoded data.

For OpenSSL 0.9.7 and later if **out* is *NULL* memory will be allocated for a buffer and the encoded data written to it. In this case **out* is not incremented and it points to the start of the data just written.

d2i_X509_bio() is similar to d2i_X509() except it attempts to parse data from BIO *bp*.

d2i_X509_fp() is similar to d2i_X509() except it attempts to parse data from FILE pointer *fp*.

i2d_X509_bio() is similar to i2d_X509() except it writes the encoding of the structure *x* to BIO *bp* and it returns 1 for success and 0 for failure.

i2d_X509_fp() is similar to i2d_X509() except it writes the encoding of the structure *x* to BIO *bp* and it returns 1 for success and 0 for failure.

NOTES

The letters *i* and *d* in for example *i2d_X509* stand for "internal" (that is an internal C structure) and "DER". So that *i2d_X509* converts from internal to DER.

The functions can also understand *BER* forms.

The actual X509 structure passed to i2d_X509() must be a valid populated *X509* structure it can *not* simply be fed with an empty structure such as that returned by X509_new().

The encoded data is in binary form and may contain embedded zeroes. Therefore any FILE pointers or BIOs should be opened in binary mode. Functions such as *strlen()* will *not* return the correct length of the encoded structure.

The ways that **in* and **out* are incremented after the operation can trap the unwary. See the *WARNINGS* section for some common errors.

The reason for the auto increment behaviour is to reflect a typical usage of ASN1 functions: after one structure is encoded or decoded another will processed after it.

EXAMPLES

Allocate and encode the DER encoding of an X509 structure:

```
 int len;
 unsigned char *buf, *p;

 len = i2d_X509(x, NULL);

 buf = OPENSSL_malloc(len);

 if (buf == NULL)
/* error */

 p = buf;

 i2d_X509(x, &p);
```

If you are using OpenSSL 0.9.7 or later then this can be simplified to:

```
 int len;
 unsigned char *buf;

 buf = NULL;

 len = i2d_X509(x, &buf);

 if (len < 0)
/* error */
```

Attempt to decode a buffer:

```
 X509 *x;

 unsigned char *buf, *p;

 int len;

 /* Something to setup buf and len */

 p = buf;

 x = d2i_X509(NULL, &p, len);

 if (x == NULL)
    /* Some error */
```

Alternative technique:

```
 X509 *x;

 unsigned char *buf, *p;

 int len;
```

```
 /* Something to setup buf and len */

 p = buf;

 x = NULL;

 if(!d2i_X509(&x, &p, len))
    /* Some error */
```

WARNINGS

The use of temporary variable is mandatory. A common mistake is to attempt to use a buffer directly as follows:

```
 int len;
 unsigned char *buf;

 len = i2d_X509(x, NULL);

 buf = OPENSSL_malloc(len);

 if (buf == NULL)
/* error */

 i2d_X509(x, &buf);

 /* Other stuff ... */

 OPENSSL_free(buf);
```

This code will result in *buf* apparently containing garbage because it was incremented after the call to point after the data just written. Also *buf* will no longer contain the pointer allocated by *OPENSSL_malloc()* and the subsequent call to *OPENSSL_free()* may well crash.

The auto allocation feature (setting buf to NULL) only works on OpenSSL 0.9.7 and later. Attempts to use it on earlier versions will typically cause a segmentation violation.

Another trap to avoid is misuse of the *xp* argument to *d2i_X509()*:

```
 X509 *x;

 if (!d2i_X509(&x, &p, len))
/* Some error */
```

This will probably crash somewhere in *d2i_X509()*. The reason for this is that the variable *x* is uninitialized and an attempt will be made to interpret its (invalid) value as an *X509* structure, typically causing a segmentation violation. If *x* is set to NULL first then this will not happen.

Restrictions

In some versions of OpenSSL the "reuse" behaviour of d2i_X509() when **px* is valid is broken and some parts of the reused structure may persist if they are not present in the new one. As a result the use of this "reuse" behaviour is strongly discouraged.

i2d_X509() will not return an error in many versions of OpenSSL, if mandatory fields are not initialized due to a programming error then the encoded structure may contain invalid data or omit the fields entirely and will not be parsed by d2i_X509(). This may be fixed in future so code should not assume that i2d_X509() will always succeed.

RETURN VALUES

d2i_X509(), d2i_X509_bio() and d2i_X509_fp() return a valid *X509* structure or *NULL* if an error occurs. The error code that can be obtained by *ERR_get_error* (3).

i2d_X509(), i2d_X509_bio() and i2d_X509_fp() return a the number of bytes successfully encoded or a negative value if an error occurs. The error code can be obtained by *ERR_get_error* (3).

i2d_X509_bio() and i2d_X509_fp() returns 1 for success and 0 if an error occurs The error code can be obtained by *ERR_get_error* (3).

SEE ALSO

ERR_get_error (3)

HISTORY

d2i_X509, i2d_X509, d2i_X509_bio, d2i_X509_fp, i2d_X509_bio and i2d_X509_fp are available in all versions of SSLeay and OpenSSL.

d2i_X509_ALGOR

NAME

d2i_X509_ALGOR, i2d_X509_ALGOR – AlgorithmIdentifier functions.

Synopsis

```
#include <openssl/x509.h>
X509_ALGOR *d2i_X509_ALGOR(X509_ALGOR **a, unsigned char **pp, long length);
int i2d_X509_ALGOR(X509_ALGOR *a, unsigned char **pp);
```

DESCRIPTION

These functions decode and encode an *X509_ALGOR* structure which is equivalent to the *AlgorithmIdentifier* structure.

Othewise these behave in a similar way to d2i_X509() and i2d_X509() described in the `d2i_X509(3)` manual page.

SEE ALSO

`d2i_X509(3)`

HISTORY

None.

d2i_X509_CRL

NAME

d2i_X509_CRL, i2d_X509_CRL, d2i_X509_CRL_bio, d2i_509_CRL_fp, i2d_X509_CRL_bio, i2d_X509_CRL_fp – PKCS#10 certificate request functions.

Synopsis

```
#include <openssl/x509.h>
X509_CRL *d2i_X509_CRL(X509_CRL **a, unsigned char **pp, long length);
int i2d_X509_CRL(X509_CRL *a, unsigned char **pp);
X509_CRL *d2i_X509_CRL_bio(BIO *bp, X509_CRL **x);
X509_CRL *d2i_X509_CRL_fp(FILE *fp, X509_CRL **x);
int i2d_X509_CRL_bio(X509_CRL *x, BIO *bp);
int i2d_X509_CRL_fp(X509_CRL *x, FILE *fp);
```

DESCRIPTION

These functions decode and encode an X509 CRL (certificate revocation list).

Othewise the functions behave in a similar way to d2i_X509() and i2d_X509() described in the `d2i_X509(3)` manual page.

SEE ALSO

`d2i_X509(3)`

HISTORY

None.

d2i_X509_NAME

NAME

d2i_X509_NAME, i2d_X509_NAME – X509_NAME encoding functions

Synopsis

```
#include <openssl/x509.h>
X509_NAME *d2i_X509_NAME(X509_NAME **a, unsigned char **pp, long length);
int i2d_X509_NAME(X509_NAME *a, unsigned char **pp);
```

DESCRIPTION

These functions decode and encode an *X509_NAME* structure which is the the same as the *Name* type defined in RFC2459 (and elsewhere) and used for example in certificate subject and issuer names.

Othewise the functions behave in a similar way to d2i_X509() and i2d_X509() described in the `d2i_X509(3)` manual page.

SEE ALSO

`d2i_X509(3)`

HISTORY

None.

d2i_X509_REQ

NAME

d2i_X509_REQ, i2d_X509_REQ, d2i_X509_REQ_bio, d2i_X509_REQ_fp, i2d_X509_REQ_bio, i2d_X509_REQ_fp – PKCS#10 certificate request functions.

Synopsis

```
#include <openssl/x509.h>
X509_REQ *d2i_X509_REQ(X509_REQ **a, unsigned char **pp, long length);
int i2d_X509_REQ(X509_REQ *a, unsigned char **pp);
X509_REQ *d2i_X509_REQ_bio(BIO *bp, X509_REQ **x);
X509_REQ *d2i_X509_REQ_fp(FILE *fp, X509_REQ **x);
int i2d_X509_REQ_bio(X509_REQ *x, BIO *bp);
int i2d_X509_REQ_fp(X509_REQ *x, FILE *fp);
```

DESCRIPTION

These functions decode and encode a PKCS#10 certificate request.

Othewise these behave in a similar way to d2i_X509() and i2d_X509() described in the `d2i_X509(3)` manual page.

SEE ALSO

`d2i_X509(3)`

HISTORY

None.

d2i_X509_SIG

NAME

d2i_X509_SIG, i2d_X509_SIG – DigestInfo functions.

Synopsis

```
#include <openssl/x509.h>
X509_SIG *d2i_X509_SIG(X509_SIG **a, unsigned char **pp, long length);
int i2d_X509_SIG(X509_SIG *a, unsigned char **pp);
```

DESCRIPTION

These functions decode and encode an X509_SIG structure which is equivalent to the *DigestInfo* structure defined in PKCS#1 and PKCS#7.

Othewise these behave in a similar way to d2i_X509() and i2d_X509() described in the `d2i_X509(3)` manual page.

SEE ALSO

`d2i_X509(3)`

HISTORY

None.

DES_random_key

NAME

DES_random_key, DES_set_key, DES_key_sched, 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_cbc_cksum, DES_quad_cksum, DES_string_to_key, DES_string_to_2keys, DES_fcrypt, DES_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);
nt 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);
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);
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 is 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.

DES_random_key() generates a random key. The PRNG must be seeded prior to using this function (see *rand* (3)). If the PRNG could not generate a secure key, 0 is returned.

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() function.

DES_set_key_checked() will check that the key passed is of odd parity and is not a week 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.

DES_set_key() works like DES_set_key_checked() if the *DES_check_key* flag is non-zero, otherwise like DES_set_key_unchecked(). These functions are available for compatibility; it is recommended to use a function that does not depend on a global variable.

DES_set_odd_parity() sets the parity of the passed *key* to odd.

DES_is_weak_key() 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, so it is not really worth checking for them.

The following routines mostly operate on an input and output stream of *DES_cblock*s.

DES_ecb_encrypt() 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. DES_ecb_encrypt() does not return a value.

DES_ecb3_encrypt() encrypts/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 of if *ks1*, *ks2* and *ks3* are the same, it is equivalent to just encryption using ECB mode and *ks1* as the key.

The macro DES_ecb2_encrypt() is provided to perform two-key Triple-DES encryption by using *ks1* for the final encryption.

DES_ncbc_encrypt() encrypts/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.

DES_xcbc_encrypt() is RSA's DESX mode of DES. It uses *inw* and *outw* to 'whiten' the encryption. *inw* and *outw* are secret (unlike the iv) and are as such, part of the key. So the key is sort of 24 bytes. This is much better than CBC DES.

DES_ede3_cbc_encrypt() 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.

DES_pcbc_encrypt() encrypt/decrypts using the propagating cipher block chaining mode used by Kerberos v4. Its parameters are the same as DES_ncbc_encrypt().

DES_cfb_encrypt() encrypt/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. Note: 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.

DES_cfb64_encrypt() implements CFB mode of DES with 64bit feedback. Why is this useful you ask? 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. num contains 'how far' we are though ivec. If this does not make much sense, read more about cfb mode of DES :-).

DES_ede3_cfb64_encrypt() and DES_ede2_cfb64_encrypt() is the same as DES_cfb64_encrypt() except that Triple-DES is used.

DES_ofb_encrypt() 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. Note: 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.

DES_ofb64_encrypt() is the same as DES_cfb64_encrypt() using Output Feed Back mode.

DES_ede3_ofb64_encrypt() and DES_ede2_ofb64_encrypt() is 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.

DES_cbc_cksum() 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* (3) etc. instead.

DES_quad_cksum() 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:

DES_fcrypt() is a fast version of the Unix *crypt* (3) function. This version takes only a small amount of space relative to other fast crypt() implementations. This is different to 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.

DES_crypt() 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* (3).

DES_enc_write() 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.

DES_enc_read() 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.

Warning: 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. So don't use these functions unless you are sure you know what you do (in which case you might not want to use them anyway). They cannot handle non-blocking sockets. DES_enc_read() uses an internal state and thus cannot be used on multiple files.

DES_rw_mode is used to specify the encryption mode to use with DES_enc_read() and DES_end_write(). If set to *DES_PCBC_MODE* (the default), DES_pcbc_encrypt is used. If 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* (7).

The *evp* (3) library provides higher-level encryption functions.

Restrictions

DES_3cbc_encrypt() is flawed and must not be used in applications.

DES_cbc_encrypt() does not modify *ivec*; use DES_ncbc_encrypt() instead.

DES_cfb_encrypt() and DES_ofb_encrypt() operates 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 1st 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 4th input byte. The same holds for output. This function has been implemented this way because most people will be using a multiple of 8 and because once you get into pulling bytes input bytes apart things get ugly!

DES_string_to_key() is available for backward compatibility with the MIT library. New applications should use a cryptographic hash function. The same applies for DES_string_to_2key().

CONFORMING TO

ANSI X3.106

The *des* library was written to be source code compatible with the MIT Kerberos library.

SEE ALSO

crypt (3), *des_modes* (7), *evp* (3), *rand* (3)

HISTORY

In OpenSSL 0.9.7, all des_ functions were renamed to DES_ to avoid clashes with older versions of libdes. Compatibility des_ functions are provided for a short while, as well as crypt(). Declarations for these are in <openssl/des_old.h>. There is no DES_ variant for des_random_seed(). This will happen to other functions as well if they are deemed redundant (des_random_seed() just calls RAND_seed() and is present for backward compatibility only), buggy or already scheduled for removal.

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() and des_string_to_key() are available in the MIT Kerberos library; des_check_key_parity(), des_fixup_key_parity() and des_is_weak_key() are available in newer versions of that library.

des_set_key_checked() and des_set_key_unchecked() were added in OpenSSL 0.9.5.

des_generate_random_block(), des_init_random_number_generator(), des_new_random_key(), des_set_random_generator_seed() and des_set_sequence_number() and des_rand_data() are used in newer versions of Kerberos but are not implemented here.

des_random_key() generated cryptographically weak random data in SSLeay and in OpenSSL prior version 0.9.5, as well as in the original MIT library.

AUTHOR

Eric Young (eay@cryptsoft.com). Modified for the OpenSSL project (http://www.openssl.org).

Modes

NAME

Modes, of, DES – the 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.

OVERVIEW

Electronic Codebook Mode (ECB)

Normally, this is found as the function *algorithm*_ecb_encrypt().

- 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 function *algorithm*_cbc_encrypt(). Be aware that des_cbc_encrypt() is not really DES CBC (it does not update the IV); use des_ncbc_encrypt() 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 function *algorithm*_cfb_encrypt().

- 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). In my implementation this is always the case.

- 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 function *algorithm*_ofb_encrypt().

- 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 re-initialized.
- Each re-initialization 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 function *algorithm*_ecb3_encrypt().

- 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 effective key strength to only slightly more than 56 bits, but these require a lot of memory.
- If all 3 keys are the same, this is effectively the same as normal ecb mode.

Triple CBC Mode

Normally, this is found as the function *algorithm*_ede3_cbc_encrypt().

- 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.

NOTES

This text was been written in large parts 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

blowfish (3), *des* (3), *idea* (3), `rc2(3)`

dh

NAME

dh – Diffie-Hellman key agreement

Synopsis

```
#include <openssl/dh.h>
#include <openssl/engine.h> DH *DH_new(void);
voidDH_free(DH *dh); intDH_size(const DH *dh);
DH *DH_generate_parameters(int prime_len, int generator, void (*callback)(int, int, void
*), void *cb_arg); intDH_check(const DH *dh, int *codes);
intDH_generate_key(DH *dh); intDH_compute_key(unsigned char *key, BIGNUM *pub_key, DH
*dh); void DH_set_default_method(const DH_METHOD *meth);
const DH_METHOD *DH_get_default_method(void);
int DH_set_method(DH *dh, const DH_METHOD *meth);
DH *DH_new_method(ENGINE *engine);
const DH_METHOD *DH_OpenSSL(void);
int DH_get_ex_new_index(long argl, char *argp, int (*new_func)(), int (*dup_func)(), void
(*free_func)());
int DH_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(const DH *a, unsigned char **pp);
intDHparams_print_fp(FILE *fp, const DH *x);
intDHparams_print(BIO *bp, const DH *x);
```

DESCRIPTION

These functions implement the Diffie-Hellman key agreement protocol. The generation of shared DH parameters is described in *DH_generate_parameters* (3); *DH_generate_key* (3) describes 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
```

Note that DH keys may use non-standard *DH_METHOD* implementations, either directly or by the use of *ENGINE* modules. In some cases (eg. an ENGINE providing support for hardware-embedded keys), these BIGNUM values will not be used by the implementation or may be used for alternative data storage. For this reason, applications should generally avoid using DH structure elements directly and instead use API functions to query or modify keys.

SEE ALSO

dhparam (1), *bn* (3), *dsa* (3), *err* (3), *rand* (3), *rsa* (3), *engine* (3), *DH_set_method* (3), *DH_new* (3), *DH_get_ex_new_index* (3), *DH_generate_parameters* (3), *DH_compute_key* (3), d2*i_DHparams* (3), *RSA_print* (3)

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

DH_generate_key() 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.

DH_generate_key() 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.

DH_compute_key() 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*. *key* must point to *DH_size(dh)* bytes of memory.

RETURN VALUES

DH_generate_key() returns 1 on success, 0 otherwise.

DH_compute_key() returns the size of the shared secret on success, -1 on error.

The error codes can be obtained by *ERR_get_error* (3).

SEE ALSO

dh (3), *ERR_get_error* (3), *rand* (3), *DH_size* (3)

HISTORY

DH_generate_key() and DH_compute_key() are available in all versions of SSLeay and OpenSSL.

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

DH_generate_parameters() 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().

prime_len is the length in bits of the safe prime to be generated. *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* (3) 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, i.e. it does not equal 2 or 5.

RETURN VALUES

DH_generate_parameters() returns a pointer to the DH structure, or NULL if the parameter generation fails. The error codes can be obtained by *ERR_get_error* (3).

DH_check() returns 1 if the check could be performed, 0 otherwise.

NOTES

DH_generate_parameters() 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.

SEE ALSO

dh (3), *ERR_get_error* (3), *rand* (3), *DH_free* (3)

HISTORY

DH_check() 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.

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* (3).

SEE ALSO

dh (3)

HISTORY

DH_get_ex_new_index(), DH_set_ex_data() and DH_get_ex_data() are available since OpenSSL 0.9.5.

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

DH_new() allocates and initializes a *DH* structure.

DH_free() 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, DH_new() returns *NULL* and sets an error code that can be obtained by *ERR_get_error* (3). Otherwise it returns a pointer to the newly allocated structure.

DH_free() returns no value.

SEE ALSO

dh (3), *ERR_get_error* (3), *DH_generate_parameters* (3), *DH_generate_key* (3)

HISTORY

DH_new() and DH_free() are available in all versions of SSLeay and OpenSSL.

DH_set_default_method

NAME

DH_set_default_method, DH_get_default_method, DH_set_method, DH_new_method, DH_OpenSSL – select DH method

Synopsis

```
#include <openssl/dh.h>
#include <openssl/engine.h>
void DH_set_default_method(const DH_METHOD *meth);
const DH_METHOD *DH_get_default_method(void);
int DH_set_method(DH *dh, const DH_METHOD *meth);
DH *DH_new_method(ENGINE *engine);
const 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. IMPORTANT: See the NOTES section for important information about how these DH API functions are affected by the use of *ENGINE* API calls.

Initially, the default DH_METHOD is the OpenSSL internal implementation, as returned by DH_OpenSSL().

DH_set_default_method() makes *meth* the default method for all DH structures created later. *NB*: This is true only whilst no ENGINE has been set as a default for DH, so this function is no longer recommended.

DH_get_default_method() returns a pointer to the current default DH_METHOD. However, the meaningfulness of this result is dependant on whether the ENGINE API is being used, so this function is no longer recommended.

DH_set_method() selects *meth* to perform all operations using the key *dh*. This will replace the DH_METHOD used by the DH key and if the previous method was supplied by an ENGINE, the handle to that ENGINE will be released during the change. It is possible to have DH keys that only work with certain DH_METHOD implementations (eg. from an ENGINE module that supports embedded hardware-protected keys), and in such cases attempting to change the DH_METHOD for the key can have unexpected results.

DH_new_method() allocates and initializes a DH structure so that *engine* will be used for the DH operations. If *engine* is NULL, the default ENGINE for DH operations is used, and if no default ENGINE is set, the DH_METHOD controlled by DH_set_default_method() is used.

THE 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

DH_OpenSSL() and DH_get_default_method() return pointers to the respective *DH_METHOD*s.

DH_set_default_method() returns no value.

DH_set_method() returns non-zero if the provided *meth* was successfully set as the method for *dh* (including unloading the ENGINE handle if the previous method was supplied by an ENGINE).

DH_new_method() returns NULL and sets an error code that can be obtained by *ERR_get_error* (3) if the allocation fails. Otherwise it returns a pointer to the newly allocated structure.

NOTES

As of version 0.9.7, DH_METHOD implementations are grouped together with other algorithmic APIs (eg. RSA_METHOD, EVP_CIPHER, etc) in *ENGINE* modules. If a default ENGINE is specified for DH functionality using an ENGINE API function, that will override any DH defaults set using the DH API (ie. DH_set_default_method()). For this reason, the ENGINE API is the recommended way to control default implementations for use in DH and other cryptographic algorithms.

SEE ALSO

dh (3), *DH_new* (3)

HISTORY

DH_set_default_method(), DH_get_default_method(), DH_set_method(), DH_new_method() and DH_OpenSSL() were added in OpenSSL 0.9.4.

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 the engine version of OpenSSL 0.9.6. For 0.9.7, the handling of defaults in the ENGINE API was restructured so that this change was reversed, and behaviour of the other functions resembled more closely the previous behaviour. The behaviour of defaults in the ENGINE API now transparently overrides the behaviour of defaults in the DH API without requiring changing these function prototypes.

DH_size

NAME

DH_size – get Diffie-Hellman prime size

Synopsis

```
#include <openssl/dh.h>
int DH_size(DH *dh);
```

DESCRIPTION

This 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().

dh->p must not be *NULL*.

RETURN VALUE

The size in bytes.

SEE ALSO

dh (3), *DH_generate_key* (3)

HISTORY

DH_size() is available in all versions of SSLeay and OpenSSL.

dsa

NAME

dsa – Digital Signature Algorithm

Synopsis

```
#include <openssl/dsa.h>
#include <openssl/engine.h>
DSA *DSA_new(void);
voidDSA_free(DSA *dsa);
intDSA_size(const 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(const 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, const unsigned char *sigbuf,
int siglen, DSA *dsa);
void DSA_set_default_method(const DSA_METHOD *meth);
const DSA_METHOD *DSA_get_default_method(void);
int DSA_set_method(DSA *dsa, const DSA_METHOD *meth);
DSA *DSA_new_method(ENGINE *engine);
const 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(const 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(const 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(const DSA *a, unsigned char **pp);
int i2d_DSAPrivateKey(const DSA *a, unsigned char **pp);
inti2d_DSAparams(const DSA *a,unsigned char **pp);
intDSAparams_print(BIO *bp, const DSA *x);
intDSAparams_print_fp(FILE *fp, const DSA *x);
intDSA_print(BIO *bp, const DSA *x, int off);
intDSA_print_fp(FILE *bp, const 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* (3); *DSA_generate_key* (3) describes how to generate a signature key. Signature generation and verification are described in *DSA_sign* (3).

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.

Note that DSA keys may use non-standard *DSA_METHOD* implementations, either directly or by the use of *ENGINE* modules. In some cases (eg. an ENGINE providing support for hardware-embedded keys), these BIGNUM values will not be used by the implementation or may be used for alternative data storage. For this reason, applications should generally avoid using DSA structure elements directly and instead use API functions to query or modify keys.

CONFORMING TO

US Federal Information Processing Standard FIPS 186 (Digital Signature Standard, DSS), ANSI X9.30

SEE ALSO

bn (3), *dh* (3), *err* (3), *rand* (3), *rsa* (3), *sha* (3), *engine* (3), *DSA_new* (3), *DSA_size* (3), *DSA_generate_parameters* (3), *DSA_dup_DH* (3), *DSA_generate_key* (3), *DSA_sign* (3), *DSA_set_method* (3), *DSA_get_ex_new_index* (3), *RSA_print* (3)

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

DSA_do_sign() 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.

DSA_sign_setup (3) may be used to precompute part of the signing operation in case signature generation is time-critical.

DSA_do_verify() verifies that the signature *sig* matches a given message digest *dgst* of size *len*. *dsa* is the signer's public key.

RETURN VALUES

DSA_do_sign() returns the signature, NULL on error. DSA_do_verify() returns 1 for a valid signature, 0 for an incorrect signature and -1 on error. The error codes can be obtained by *ERR_get_error* (3).

SEE ALSO

dsa (3), *ERR_get_error* (3), *rand* (3), *DSA_SIG_new* (3), *DSA_sign* (3)

HISTORY

DSA_do_sign() and DSA_do_verify() were added in OpenSSL 0.9.3.

DSA_dup_DH

NAME

DSA_dup_DH – create a DH structure out of DSA structure

Synopsis

```
#include <openssl/dsa.h>
DH * DSA_dup_DH(const DSA *r);
```

DESCRIPTION

DSA_dup_DH() duplicates DSA parameters/keys as DH parameters/keys. q is lost during that conversion, but the resulting DH parameters contain its length.

RETURN VALUE

DSA_dup_DH() returns the new *DH* structure, and NULL on error. The error codes can be obtained by *ERR_get_error* (3).

NOTE

Be careful to avoid small subgroup attacks when using this.

SEE ALSO

dh (3), *dsa* (3), *ERR_get_error* (3)

HISTORY

DSA_dup_DH() was added in OpenSSL 0.9.4.

DSA_generate_key

NAME

DSA_generate_key – generate DSA key pair

Synopsis

```
#include <openssl/dsa.h>
int DSA_generate_key(DSA *a);
```

DESCRIPTION

DSA_generate_key() 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

DSA_generate_key() returns 1 on success, 0 otherwise. The error codes can be obtained by *ERR_get_error* (3).

SEE ALSO

dsa (3), *ERR_get_error* (3), *rand* (3), *DSA_generate_parameters* (3)

HISTORY

DSA_generate_key() is available since SSLeay 0.8.

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

DSA_generate_parameters() generates primes p and q and a generator g for use in the DSA.

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*.

DSA_generate_parameters() 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.

RETURN VALUE

DSA_generate_parameters() returns a pointer to the DSA structure, or *NULL* if the parameter generation fails. The error codes can be obtained by *ERR_get_error* (3).

Restrictions

Seed lengths > 20 are not supported.

SEE ALSO

dsa (3), *ERR_get_error* (3), *rand* (3), *DSA_free* (3)

HISTORY

DSA_generate_parameters() 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* (3), i.e. once for each witness.

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* (3).

SEE ALSO

dsa (3)

HISTORY

DSA_get_ex_new_index(), DSA_set_ex_data() and DSA_get_ex_data() are available since OpenSSL 0.9.5.

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

DSA_new() allocates and initializes a *DSA* structure. It is equivalent to calling DSA_new_method(NULL).

DSA_free() 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, DSA_new() returns *NULL* and sets an error code that can be obtained by *ERR_get_error* (3). Otherwise it returns a pointer to the newly allocated structure.

DSA_free() returns no value.

SEE ALSO

dsa (3), *ERR_get_error* (3), *DSA_generate_parameters* (3), *DSA_generate_key* (3)

HISTORY

DSA_new() and DSA_free() are available in all versions of SSLeay and OpenSSL.

DSA_set_default_method

NAME

DSA_set_default_method, DSA_get_default_method, DSA_set_method, DSA_new_method, DSA_OpenSSL – select DSA method

Synopsis

```
#include <openssl/dsa.h>
#include <openssl/engine.h>
void DSA_set_default_method(const DSA_METHOD *meth);
const DSA_METHOD *DSA_get_default_method(void);
int DSA_set_method(DSA *dsa, const DSA_METHOD *meth);
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 may be used. IMPORTANT: See the NOTES section for important information about how these DSA API functions are affected by the use of *ENGINE* API calls.

Initially, the default DSA_METHOD is the OpenSSL internal implementation, as returned by DSA_OpenSSL().

DSA_set_default_method() makes *meth* the default method for all DSA structures created later. *NB*: This is true only whilst no ENGINE has been set as a default for DSA, so this function is no longer recommended.

DSA_get_default_method() returns a pointer to the current default DSA_METHOD. However, the meaningfulness of this result is dependant on whether the ENGINE API is being used, so this function is no longer recommended.

DSA_set_method() selects *meth* to perform all operations using the key *rsa*. This will replace the DSA_METHOD used by the DSA key and if the previous method was supplied by an ENGINE, the handle to that ENGINE will be released during the change. It is possible to have DSA keys that only work with certain DSA_METHOD implementations (eg. from an ENGINE module that supports embedded hardware-protected keys), and in such cases attempting to change the DSA_METHOD for the key can have unexpected results.

DSA_new_method() 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, and if no default ENGINE is set, the DSA_METHOD controlled by DSA_set_default_method() 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

DSA_OpenSSL() and DSA_get_default_method() return pointers to the respective *DSA_METHOD*s.

DSA_set_default_method() returns no value.

DSA_set_method() returns non-zero if the provided *meth* was successfully set as the method for *dsa* (including unloading the ENGINE handle if the previous method was supplied by an ENGINE).

DSA_new_method() returns NULL and sets an error code that can be obtained by *ERR_get_error* (3) if the allocation fails. Otherwise it returns a pointer to the newly allocated structure.

NOTES

As of version 0.9.7, DSA_METHOD implementations are grouped together with other algorithmic APIs (eg. RSA_METHOD, EVP_CIPHER, etc) in *ENGINE* modules. If a default ENGINE is specified for DSA functionality using an ENGINE API function, that will override any DSA defaults set using the DSA API (ie. DSA_set_default_method()). For this reason, the ENGINE API is the recommended way to control default implementations for use in DSA and other cryptographic algorithms.

SEE ALSO

dsa (3), *DSA_new* (3)

HISTORY

DSA_set_default_method(), DSA_get_default_method(), DSA_set_method(), DSA_new_method() and DSA_OpenSSL() were added in OpenSSL 0.9.4.

DSA_set_default_openssl_method() and DSA_get_default_openssl_method() replaced DSA_set_default_method() and DSA_get_default_method() respectively, and DSA_set_method() and DSA_new_method() were altered to use *ENGINE*s rather than *DSA_METHOD*s during development of the engine version of OpenSSL 0.9.6. For 0.9.7, the handling of defaults in the ENGINE API was restructured so that this change was reversed, and behaviour of the other functions resembled more closely the previous behaviour. The behaviour of defaults in the ENGINE API now transparently overrides the behaviour of defaults in the DSA API without requiring changing these function prototypes.

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);
voidDSA_SIG_free(DSA_SIG *a);
```

DESCRIPTION

DSA_SIG_new() allocates and initializes a *DSA_SIG* structure.

DSA_SIG_free() 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 by *ERR_get_error* (3). Otherwise it returns a pointer to the newly allocated structure.

DSA_SIG_free() returns no value.

SEE ALSO

dsa (3), *ERR_get_error* (3), *DSA_do_sign* (3)

HISTORY

DSA_SIG_new() and DSA_SIG_free() were added in OpenSSL 0.9.3.

DSA_sign

NAME

DSA_sign, DSA_sign_setup, DSA_verify – DSA signatures

Synopsis

```
#include <openssl/dsa.h>
intDSA_sign(int type, 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 type, const unsigned char *dgst, int len, unsigned char *sigbuf, int
siglen, DSA *dsa);
```

DESCRIPTION

DSA_sign() 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 places in **siglen*. *sigret* must point to DSA_size(*dsa*) bytes of memory.

DSA_sign_setup() 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*. *ctx* is a pre-allocated *BN_CTX* or NULL.

DSA_verify() verifies that the signature *sigbuf* of size *siglen* matches a given message digest *dgst* of size *len*. *dsa* is the signer's public key.

The *type* parameter is ignored.

The PRNG must be seeded before DSA_sign() (or DSA_sign_setup()) is called.

RETURN VALUES

DSA_sign() and DSA_sign_setup() return 1 on success, 0 on error. DSA_verify() returns 1 for a valid signature, 0 for an incorrect signature and -1 on error. The error codes can be obtained by *ERR_get_error* (3).

CONFORMING TO

US Federal Information Processing Standard FIPS 186 (Digital Signature Standard, DSS), ANSI X9.30

SEE ALSO

dsa (3), *ERR_get_error* (3), *rand* (3), *DSA_do_sign* (3)

HISTORY

DSA_sign() and DSA_verify() are available in all versions of SSLeay. DSA_sign_setup() was added in SSLeay 0.8.

DSA_size

NAME

DSA_size – get DSA signature size

Synopsis

```
#include <openssl/dsa.h>
int DSA_size(const DSA *dsa);
```

DESCRIPTION

This 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.

dsa->q must not be *NULL*.

RETURN VALUE

The size in bytes.

SEE ALSO

dsa (3), *DSA_sign* (3)

HISTORY

DSA_size() is available in all versions of SSLeay and OpenSSL.

engine

NAME

engine – ENGINE cryptographic module support

Synopsis

```
#include <openssl/engine.h>
ENGINE *ENGINE_get_first(void);
ENGINE *ENGINE_get_last(void);
ENGINE *ENGINE_get_next(ENGINE *e);
ENGINE *ENGINE_get_prev(ENGINE *e);
int ENGINE_add(ENGINE *e);
int ENGINE_remove(ENGINE *e);
ENGINE *ENGINE_by_id(const char *id);
int ENGINE_init(ENGINE *e);
int ENGINE_finish(ENGINE *e);
void ENGINE_load_openssl(void);
void ENGINE_load_dynamic(void);
void ENGINE_load_cswift(void);
void ENGINE_load_chil(void);
void ENGINE_load_atalla(void);
void ENGINE_load_nuron(void);
void ENGINE_load_ubsec(void);
void ENGINE_load_aep(void);
void ENGINE_load_sureware(void);
void ENGINE_load_4758cca(void);
void ENGINE_load_openbsd_dev_crypto(void);
void ENGINE_load_builtin_engines(void);
void ENGINE_cleanup(void);
ENGINE *ENGINE_get_default_RSA(void);
ENGINE *ENGINE_get_default_DSA(void);
ENGINE *ENGINE_get_default_DH(void);
ENGINE *ENGINE_get_default_RAND(void);
ENGINE *ENGINE_get_cipher_engine(int nid);
ENGINE *ENGINE_get_digest_engine(int nid);
int ENGINE_set_default_RSA(ENGINE *e);
int ENGINE_set_default_DSA(ENGINE *e);
int ENGINE_set_default_DH(ENGINE *e);
int ENGINE_set_default_RAND(ENGINE *e);
int ENGINE_set_default_ciphers(ENGINE *e);
int ENGINE_set_default_digests(ENGINE *e);
int ENGINE_set_default_string(ENGINE *e, const char *list);
int ENGINE_set_default(ENGINE *e, unsigned int flags);
unsigned int ENGINE_get_table_flags(void);
void ENGINE_set_table_flags(unsigned int flags);
int ENGINE_register_RSA(ENGINE *e);
void ENGINE_unregister_RSA(ENGINE *e);
void ENGINE_register_all_RSA(void);
int ENGINE_register_DSA(ENGINE *e);
void ENGINE_unregister_DSA(ENGINE *e);
void ENGINE_register_all_DSA(void);
```

```
int ENGINE_register_DH(ENGINE *e);
void ENGINE_unregister_DH(ENGINE *e);
void ENGINE_register_all_DH(void);
int ENGINE_register_RAND(ENGINE *e);
void ENGINE_unregister_RAND(ENGINE *e);
void ENGINE_register_all_RAND(void);
int ENGINE_register_ciphers(ENGINE *e);
void ENGINE_unregister_ciphers(ENGINE *e);
void ENGINE_register_all_ciphers(void);
int ENGINE_register_digests(ENGINE *e);
void ENGINE_unregister_digests(ENGINE *e);
void ENGINE_register_all_digests(void);
int ENGINE_register_complete(ENGINE *e);
int ENGINE_register_all_complete(void);
int ENGINE_ctrl(ENGINE *e, int cmd, long i, void *p, void (*f)());
int ENGINE_cmd_is_executable(ENGINE *e, int cmd);
int ENGINE_ctrl_cmd(ENGINE *e, const char *cmd_name, long i, void *p, void (*f)(), int
cmd_optional);
int ENGINE_ctrl_cmd_string(ENGINE *e, const char *cmd_name, const char *arg, int
cmd_optional);
int ENGINE_set_ex_data(ENGINE *e, int idx, void *arg);
void *ENGINE_get_ex_data(const ENGINE *e, int idx);
int ENGINE_get_ex_new_index(long argl, void *argp, CRYPTO_EX_new *new_func, CRYPTO_EX_dup
*dup_func, CRYPTO_EX_free *free_func);
ENGINE *ENGINE_new(void);
int ENGINE_free(ENGINE *e);
int ENGINE_set_id(ENGINE *e, const char *id);
int ENGINE_set_name(ENGINE *e, const char *name);
int ENGINE_set_RSA(ENGINE *e, const RSA_METHOD *rsa_meth);
int ENGINE_set_DSA(ENGINE *e, const DSA_METHOD *dsa_meth);
int ENGINE_set_DH(ENGINE *e, const DH_METHOD *dh_meth);
int ENGINE_set_RAND(ENGINE *e, const RAND_METHOD *rand_meth);
int ENGINE_set_destroy_function(ENGINE *e, ENGINE_GEN_INT_FUNC_PTR destroy_f);
int ENGINE_set_init_function(ENGINE *e, ENGINE_GEN_INT_FUNC_PTR init_f);
int ENGINE_set_finish_function(ENGINE *e, ENGINE_GEN_INT_FUNC_PTR finish_f);
int ENGINE_set_ctrl_function(ENGINE *e, ENGINE_CTRL_FUNC_PTR ctrl_f);
int ENGINE_set_load_privkey_function(ENGINE *e, ENGINE_LOAD_KEY_PTR loadpriv_f);
int ENGINE_set_load_pubkey_function(ENGINE *e, ENGINE_LOAD_KEY_PTR loadpub_f);
int ENGINE_set_ciphers(ENGINE *e, ENGINE_CIPHERS_PTR f);
int ENGINE_set_digests(ENGINE *e, ENGINE_DIGESTS_PTR f);
int ENGINE_set_flags(ENGINE *e, int flags);
int ENGINE_set_cmd_defns(ENGINE *e, const ENGINE_CMD_DEFN *defns);
const char *ENGINE_get_id(const ENGINE *e); c
onst char *ENGINE_get_name(const ENGINE *e);
const RSA_METHOD *ENGINE_get_RSA(const ENGINE *e);
const DSA_METHOD *ENGINE_get_DSA(const ENGINE *e);
const DH_METHOD *ENGINE_get_DH(const ENGINE *e);
const RAND_METHOD *ENGINE_get_RAND(const ENGINE *e);
ENGINE_GEN_INT_FUNC_PTR ENGINE_get_destroy_function(const ENGINE *e);
ENGINE_GEN_INT_FUNC_PTR ENGINE_get_init_function(const ENGINE *e);
ENGINE_GEN_INT_FUNC_PTR ENGINE_get_finish_function(const ENGINE *e);
ENGINE_CTRL_FUNC_PTR ENGINE_get_ctrl_function(const ENGINE *e);
ENGINE_LOAD_KEY_PTR ENGINE_get_load_privkey_function(const ENGINE *e);
```

```
ENGINE_LOAD_KEY_PTR ENGINE_get_load_pubkey_function(const ENGINE *e);
ENGINE_CIPHERS_PTR ENGINE_get_ciphers(const ENGINE *e);
ENGINE_DIGESTS_PTR ENGINE_get_digests(const ENGINE *e);
const EVP_CIPHER *ENGINE_get_cipher(ENGINE *e, int nid);
const EVP_MD *ENGINE_get_digest(ENGINE *e, int nid);
int ENGINE_get_flags(const ENGINE *e);
const ENGINE_CMD_DEFN *ENGINE_get_cmd_defns(const ENGINE *e);
EVP_PKEY *ENGINE_load_private_key(ENGINE *e, const char *key_id, UI_METHOD *ui_method,
void *callback_data);
EVP_PKEY *ENGINE_load_public_key(ENGINE *e, const char *key_id, UI_METHOD *ui_method, void
*callback_data); void ENGINE_add_conf_module(void);
```

DESCRIPTION

These functions create, manipulate, and use cryptographic modules in the form of *ENGINE* objects. These objects act as containers for implementations of cryptographic algorithms, and support a reference-counted mechanism to allow them to be dynamically loaded in and out of the running application.

The cryptographic functionality that can be provided by an *ENGINE* implementation includes the following abstractions;

```
RSA_METHOD - for providing alternative RSA implementations
DSA_METHOD, DH_METHOD, RAND_METHOD - alternative DSA, DH, and RAND
EVP_CIPHER - potentially multiple cipher algorithms (indexed by 'nid')
EVP_DIGEST - potentially multiple hash algorithms (indexed by 'nid')
key-loading - loading public and/or private EVP_PKEY keys
```

Reference counting and handles

Due to the modular nature of the ENGINE API, pointers to ENGINEs need to be treated as handles - ie. not only as pointers, but also as references to the underlying ENGINE object. Ie. you should obtain a new reference when making copies of an ENGINE pointer if the copies will be used (and released) independantly.

ENGINE objects have two levels of reference-counting to match the way in which the objects are used. At the most basic level, each ENGINE pointer is inherently a *structural* reference - you need a structural reference simply to refer to the pointer value at all, as this kind of reference is your guarantee that the structure can not be deallocated until you release your reference.

However, a structural reference provides no guarantee that the ENGINE has been initiliased to be usable to perform any of its cryptographic implementations - and indeed it's quite possible that most ENGINEs will not initialised at all on standard setups, as ENGINEs are typically used to support specialised hardware. To use an ENGINE's functionality, you need a *functional* reference. This kind of reference can be considered a specialised form of structural reference, because each functional reference implicitly contains a structural reference as well - however to avoid difficult-to-find programming bugs, it is recommended to treat the two kinds of reference independantly. If you have a functional reference to an ENGINE, you have a guarantee that the ENGINE has been initialised ready to perform cryptographic operations and will not be uninitialised or cleaned up until after you have released your reference.

We will discuss the two kinds of reference separately, including how to tell which one you are dealing with at any given point in time (after all they are both simply (ENGINE *) pointers, the difference is in the way they are used).

Structural references

This basic type of reference is typically used for creating new ENGINEs dynamically, iterating across OpenSSL's internal linked-list of loaded ENGINEs, reading information about an ENGINE, etc. Essentially a structural reference is sufficient if you only need to query or manipulate the data of an ENGINE implementation rather than use its functionality.

The ENGINE_new() function returns a structural reference to a new (empty) ENGINE object. Other than that, structural references come from return values to various ENGINE API functions such as; ENGINE_by_id(), ENGINE_get_first(), ENGINE_get_last(), ENGINE_get_next(), ENGINE_get_prev(). All structural references should be released by a corresponding to call to the ENGINE_free() function - the ENGINE object itself will only actually be cleaned up and deallocated when the last structural reference is released.

It should also be noted that many ENGINE API function calls that accept a structural reference will internally obtain another reference - typically this happens whenever the supplied ENGINE will be needed by OpenSSL after the function has returned. Eg. the function to add a new ENGINE to OpenSSL's internal list is ENGINE_add() - if this function returns success, then OpenSSL will have stored a new structural reference internally so the caller is still responsible for freeing their own reference with ENGINE_free() when they are finished with it. In a similar way, some functions will automatically release the structural reference passed to it if part of the function's job is to do so. Eg. the ENGINE_get_next() and ENGINE_get_prev() functions are used for iterating across the internal ENGINE list - they will return a new structural reference to the next (or previous) ENGINE in the list or NULL if at the end (or beginning) of the list, but in either case the structural reference passed to the function is released on behalf of the caller.

To clarify a particular function's handling of references, one should always consult that function's documentation "man" page, or failing that the openssl/engine.h header file includes some hints.

Functional references

As mentioned, functional references exist when the cryptographic functionality of an ENGINE is required to be available. A functional reference can be obtained in one of two ways; from an existing structural reference to the required ENGINE, or by asking OpenSSL for the default operational ENGINE for a given cryptographic purpose.

To obtain a functional reference from an existing structural reference, call the ENGINE_init() function. This returns zero if the ENGINE was not already operational and couldn't be successfully initialised (eg. lack of system drivers, no special hardware attached, etc), otherwise it will return non-zero to indicate that the ENGINE is now operational and will have allocated a new *functional* reference to the ENGINE. In this case, the supplied ENGINE pointer is, from the point of the view of the caller, both a structural reference and a functional reference - so if the caller intends to use it as a functional reference it should free the structural reference with ENGINE_free() first. If the caller wishes to use it only as a structural reference (eg. if the ENGINE_init() call was simply to test if the ENGINE seems available/online), then it should free the functional reference; all functional references are released by the ENGINE_finish() function.

The second way to get a functional reference is by asking OpenSSL for a default implementation for a given task, eg. by ENGINE_get_default_RSA(), ENGINE_get_default_cipher_engine(), etc. These are discussed in the next section, though they are not usually required by application programmers as they are used automatically when creating and using the relevant algorithm-specific types in OpenSSL, such as RSA, DSA, EVP_CIPHER_CTX, etc.

Default implementations

For each supported abstraction, the ENGINE code maintains an internal table of state to control which implementations are available for a given abstraction and which should be used by default. These implementations are registered in the tables separated-out by an 'nid' index, because abstractions like EVP_CIPHER and EVP_DIGEST support many distinct algorithms and modes - ENGINEs will support different numbers and combinations of these. In the case of other abstractions like RSA, DSA, etc, there is only one "algorithm" so all implementations implicitly register using the same 'nid' index. ENGINEs can be

registered into these tables to make themselves available for use automatically by the various abstractions, eg. RSA. For illustrative purposes, we continue with the RSA example, though all comments apply similarly to the other abstractions (they each get their own table and linkage to the corresponding section of openssl code).

When a new RSA key is being created, ie. in RSA_new_method(), a "get_default" call will be made to the ENGINE subsystem to process the RSA state table and return a functional reference to an initialised ENGINE whose RSA_METHOD should be used. If no ENGINE should (or can) be used, it will return NULL and the RSA key will operate with a NULL ENGINE handle by using the conventional RSA implementation in OpenSSL (and will from then on behave the way it used to before the ENGINE API existed - for details see *RSA_new_method* (3)).

Each state table has a flag to note whether it has processed this "get_default" query since the table was last modified, because to process this question it must iterate across all the registered ENGINEs in the table trying to initialise each of them in turn, in case one of them is operational. If it returns a functional reference to an ENGINE, it will also cache another reference to speed up processing future queries (without needing to iterate across the table). Likewise, it will cache a NULL response if no ENGINE was available so that future queries won't repeat the same iteration unless the state table changes. This behaviour can also be changed; if the ENGINE_TABLE_FLAG_NOINIT flag is set (using ENGINE_set_table_flags()), no attempted initialisations will take place, instead the only way for the state table to return a non-NULL ENGINE to the "get_default" query will be if one is expressly set in the table. Eg. ENGINE_set_default_RSA() does the same job as ENGINE_register_RSA() except that it also sets the state table's cached response for the "get_default" query.

In the case of abstractions like EVP_CIPHER, where implementations are indexed by 'nid', these flags and cached-responses are distinct for each 'nid' value.

It is worth illustrating the difference between "registration" of ENGINEs into these per-algorithm state tables and using the alternative "set_default" functions. The latter handles both "registration" and also setting the cached "default" ENGINE in each relevant state table - so registered ENGINEs will only have a chance to be initialised for use as a default if a default ENGINE wasn't already set for the same state table. Eg. if ENGINE X supports cipher nids {A,B} and RSA, ENGINE Y supports ciphers {A} and DSA, and the following code is executed;

```
ENGINE_register_complete(X);
ENGINE_set_default(Y, ENGINE_METHOD_ALL);
e1 = ENGINE_get_default_RSA();
e2 = ENGINE_get_cipher_engine(A);
e3 = ENGINE_get_cipher_engine(B);
e4 = ENGINE_get_default_DSA();
e5 = ENGINE_get_cipher_engine(C);
```

The results would be as follows;

```
assert(e1 == X);
assert(e2 == Y);
assert(e3 == X);
assert(e4 == Y);
assert(e5 == NULL);
```

Application requirements

This section will explain the basic things an application programmer should support to make the most useful elements of the ENGINE functionality available to the user. The first thing to consider is whether the programmer wishes to make alternative ENGINE modules available to the application and user. OpenSSL maintains an internal linked list of "visible" ENGINEs from which it has to operate - at start-up, this list is empty and in fact if an application does not call any ENGINE API calls and it uses static linking against

openssl, then the resulting application binary will not contain any alternative ENGINE code at all. So the first consideration is whether any/all available ENGINE implementations should be made visible to OpenSSL - this is controlled by calling the various "load" functions, eg.

```
/* Make the "dynamic" ENGINE available */
void ENGINE_load_dynamic(void);
/* Make the CryptoSwift hardware acceleration support available */
void ENGINE_load_cswift(void);
/* Make support for nCipher's "CHIL" hardware available */
void ENGINE_load_chil(void);
...
/* Make ALL ENGINE implementations bundled with OpenSSL available */
void ENGINE_load_builtin_engines(void);
```

Having called any of these functions, ENGINE objects would have been dynamically allocated and populated with these implementations and linked into OpenSSL's internal linked list. At this point it is important to mention an important API function;

```
void ENGINE_cleanup(void);
```

If no ENGINE API functions are called at all in an application, then there are no inherent memory leaks to worry about from the ENGINE functionality, however if any ENGINEs are "load"ed, even if they are never registered or used, it is necessary to use the ENGINE_cleanup() function to correspondingly cleanup before program exit, if the caller wishes to avoid memory leaks. This mechanism uses an internal callback registration table so that any ENGINE API functionality that knows it requires cleanup can register its cleanup details to be called during ENGINE_cleanup(). This approach allows ENGINE_cleanup() to clean up after any ENGINE functionality at all that your program uses, yet doesn't automatically create linker dependencies to all possible ENGINE functionality - only the cleanup callbacks required by the functionality you do use will be required by the linker.

The fact that ENGINEs are made visible to OpenSSL (and thus are linked into the program and loaded into memory at run-time) does not mean they are "registered" or called into use by OpenSSL automatically - that behaviour is something for the application to have control over. Some applications will want to allow the user to specify exactly which ENGINE they want used if any is to be used at all. Others may prefer to load all support and have OpenSSL automatically use at run-time any ENGINE that is able to successfully initialise - ie. to assume that this corresponds to acceleration hardware attached to the machine or some such thing. There are probably numerous other ways in which applications may prefer to handle things, so we will simply illustrate the consequences as they apply to a couple of simple cases and leave developers to consider these and the source code to openssl's builtin utilities as guides.

Using a specific ENGINE implementation

Here we'll assume an application has been configured by its user or admin to want to use the "ACME" ENGINE if it is available in the version of OpenSSL the application was compiled with. If it is available, it should be used by default for all RSA, DSA, and symmetric cipher operation, otherwise OpenSSL should use its builtin software as per usual. The following code illustrates how to approach this;

```
ENGINE *e;
const char *engine_id = "ACME";
ENGINE_load_builtin_engines();
e = ENGINE_by_id(engine_id);
if(!e)
    /* the engine isn't available */
    return;
if(!ENGINE_init(e)) {
    /* the engine couldn't initialise, release 'e' */
    ENGINE_free(e);
    return;
}
```

```
if(!ENGINE_set_default_RSA(e))
    /* This should only happen when 'e' can't initialise, but the previous
     * statement suggests it did. */
    abort();
ENGINE_set_default_DSA(e);
ENGINE_set_default_ciphers(e);
/* Release the functional reference from ENGINE_init() */
ENGINE_finish(e);
/* Release the structural reference from ENGINE_by_id() */
ENGINE_free(e);
```

Automatically using builtin ENGINE implementations

Here we'll assume we want to load and register all ENGINE implementations bundled with OpenSSL, such that for any cryptographic algorithm required by OpenSSL - if there is an ENGINE that implements it and can be initialise, it should be used. The following code illustrates how this can work;

```
/* Load all bundled ENGINEs into memory and make them visible */
ENGINE_load_builtin_engines();
/* Register all of them for every algorithm they collectively implement */
ENGINE_register_all_complete();
```

That's all that's required. Eg. the next time OpenSSL tries to set up an RSA key, any bundled ENGINEs that implement RSA_METHOD will be passed to ENGINE_init() and if any of those succeed, that ENGINE will be set as the default for use with RSA from then on.

Advanced configuration support

There is a mechanism supported by the ENGINE framework that allows each ENGINE implementation to define an arbitrary set of configuration "commands" and expose them to OpenSSL and any applications based on OpenSSL. This mechanism is entirely based on the use of name-value pairs and and assumes ASCII input (no unicode or UTF for now!), so it is ideal if applications want to provide a transparent way for users to provide arbitrary configuration "directives" directly to such ENGINEs. It is also possible for the application to dynamically interrogate the loaded ENGINE implementations for the names, descriptions, and input flags of their available "control commands", providing a more flexible configuration scheme. However, if the user is expected to know which ENGINE device he/she is using (in the case of specialised hardware, this goes without saying) then applications may not need to concern themselves with discovering the supported control commands and simply prefer to allow settings to passed into ENGINEs exactly as they are provided by the user.

Before illustrating how control commands work, it is worth mentioning what they are typically used for. Broadly speaking there are two uses for control commands; the first is to provide the necessary details to the implementation (which may know nothing at all specific to the host system) so that it can be initialised for use. This could include the path to any driver or config files it needs to load, required network addresses, smart-card identifiers, passwords to initialise password-protected devices, logging information, etc etc. This class of commands typically needs to be passed to an ENGINE *before* attempting to initialise it, ie. before calling ENGINE_init(). The other class of commands consist of settings or operations that tweak certain behaviour or cause certain operations to take place, and these commands may work either before or after ENGINE_init(), or in same cases both. ENGINE implementations should provide indications of this in the descriptions attached to builtin control commands and/or in external product documentation.

Issuing control commands to an ENGINE

Let's illustrate by example; a function for which the caller supplies the name of the ENGINE it wishes to use, a table of string-pairs for use before initialisation, and another table for use after initialisation. Note that the string-pairs used for control commands consist of a command "name" followed by the command "parameter" -

the parameter could be NULL in some cases but the name can not. This function should initialise the ENGINE (issuing the "pre" commands beforehand and the "post" commands afterwards) and set it as the default for everything except RAND and then return a boolean success or failure.

```
int generic_load_engine_fn(const char *engine_id,
                           const char **pre_cmds, int pre_num,
                           const char **post_cmds, int post_num)
{
    ENGINE *e = ENGINE_by_id(engine_id);
    if(!e) return 0;
    while(pre_num--) {
        if(!ENGINE_ctrl_cmd_string(e, pre_cmds[0], pre_cmds[1], 0)) {
            fprintf(stderr, "Failed command (%s - %s:%s)\n", engine_id,
                pre_cmds[0], pre_cmds[1] ? pre_cmds[1] : "(NULL)");
            ENGINE_free(e);
            return 0;
        }
pre_cmds += 2;
    }
    if(!ENGINE_init(e)) {
        fprintf(stderr, "Failed initialisation\n");
        ENGINE_free(e);
        return 0;
    }
    /* ENGINE_init() returned a functional reference, so free the structural
     * reference from ENGINE_by_id(). */
    ENGINE_free(e);
    while(post_num--) {
        if(!ENGINE_ctrl_cmd_string(e, post_cmds[0], post_cmds[1], 0)) {
            fprintf(stderr, "Failed command (%s - %s:%s)\n", engine_id,
                post_cmds[0], post_cmds[1] ? post_cmds[1] : "(NULL)");
            ENGINE_finish(e);
            return 0;
        }
post_cmds += 2;
    }
    ENGINE_set_default(e, ENGINE_METHOD_ALL & ~ENGINE_METHOD_RAND);
    /* Success */
    return 1;
}
```

Note that ENGINE_ctrl_cmd_string() accepts a boolean argument that can relax the semantics of the function - if set non-zero it will only return failure if the ENGINE supported the given command name but failed while executing it, if the ENGINE doesn't support the command name it will simply return success without doing anything. In this case we assume the user is only supplying commands specific to the given ENGINE so we set this to FALSE.

Discovering supported control commands

It is possible to discover at run-time the names, numerical-ids, descriptions and input parameters of the control commands supported from a structural reference to any ENGINE. It is first important to note that some control commands are defined by OpenSSL itself and it will intercept and handle these control commands on behalf of the ENGINE, ie. the ENGINE's ctrl() handler is not used for the control command. openssl/engine.h defines a symbol, ENGINE_CMD_BASE, that all control commands implemented by ENGINEs from. Any command value lower than this symbol is considered a "generic" command is handled directly by the OpenSSL core routines.

It is using these "core" control commands that one can discover the the control commands implemented by a given ENGINE, specifically the commands;

```
#define ENGINE_HAS_CTRL_FUNCTION10
#define ENGINE_CTRL_GET_FIRST_CMD_TYPE11
#define ENGINE_CTRL_GET_NEXT_CMD_TYPE12
#define ENGINE_CTRL_GET_CMD_FROM_NAME13
#define ENGINE_CTRL_GET_NAME_LEN_FROM_CMD14
#define ENGINE_CTRL_GET_NAME_FROM_CMD15
#define ENGINE_CTRL_GET_DESC_LEN_FROM_CMD16
#define ENGINE_CTRL_GET_DESC_FROM_CMD17
#define ENGINE_CTRL_GET_CMD_FLAGS18
```

Whilst these commands are automatically processed by the OpenSSL framework code, they use various properties exposed by each ENGINE by which to process these queries. An ENGINE has 3 properties it exposes that can affect this behaviour; it can supply a ctrl() handler, it can specify ENGINE_FLAGS_MANUAL_CMD_CTRL in the ENGINE's flags, and it can expose an array of control command descriptions. If an ENGINE specifies the ENGINE_FLAGS_MANUAL_CMD_CTRL flag, then it will simply pass all these "core" control commands directly to the ENGINE's ctrl() handler (and thus, it must have supplied one), so it is up to the ENGINE to reply to these "discovery" commands itself. If that flag is not set, then the OpenSSL framework code will work with the following rules;

```
if no ctrl() handler supplied;
    ENGINE_HAS_CTRL_FUNCTION returns FALSE (zero),
    all other commands fail.
if a ctrl() handler was supplied but no array of control commands;
    ENGINE_HAS_CTRL_FUNCTION returns TRUE,
    all other commands fail.
if a ctrl() handler and array of control commands was supplied;
    ENGINE_HAS_CTRL_FUNCTION returns TRUE,
    all other commands proceed processing ...
```

If the ENGINE's array of control commands is empty then all other commands will fail, otherwise; ENGINE_CTRL_GET_FIRST_CMD_TYPE returns the identifier of the first command supported by the ENGINE, ENGINE_GET_NEXT_CMD_TYPE takes the identifier of a command supported by the ENGINE and returns the next command identifier or fails if there are no more, ENGINE_CMD_FROM_NAME takes a string name for a command and returns the corresponding identifier or fails if no such command name exists, and the remaining commands take a command identifier and return properties of the corresponding commands. All except ENGINE_CTRL_GET_FLAGS return the string length of a command name or description, or populate a supplied character buffer with a copy of the command name or description. ENGINE_CTRL_GET_FLAGS returns a bitwise-OR'd mask of the following possible values;

```
#define ENGINE_CMD_FLAG_NUMERIC(unsigned int)0x0001
#define ENGINE_CMD_FLAG_STRING(unsigned int)0x0002
#define ENGINE_CMD_FLAG_NO_INPUT(unsigned int)0x0004
#define ENGINE_CMD_FLAG_INTERNAL(unsigned int)0x0008
```

If the ENGINE_CMD_FLAG_INTERNAL flag is set, then any other flags are purely informational to the caller - this flag will prevent the command being usable for any higher-level ENGINE functions such as ENGINE_ctrl_cmd_string(). "INTERNAL" commands are not intended to be exposed to text-based configuration by applications, administrations, users, etc. These can support arbitrary operations via ENGINE_ctrl(), including passing to and/or from the control commands data of any arbitrary type. These commands are supported in the discovery mechanisms simply to allow applications determinie if an ENGINE supports certain specific commands it might want to use (eg. application "foo" might query various ENGINEs to see if they implement "FOO_GET_VENDOR_LOGO_GIF" - and ENGINE could therefore decide whether or not to support this "foo"-specific extension).

Future developments

The ENGINE API and internal architecture is currently being reviewed. Slated for possible release in 0.9.8 is support for transparent loading of "dynamic" ENGINEs (built as self-contained shared-libraries). This would allow ENGINE implementations to be provided independantly of OpenSSL libraries and/or OpenSSL-based applications, and would also remove any requirement for applications to explicitly use the "dynamic" ENGINE to bind to shared-library implementations.

SEE ALSO

rsa (3), *dsa* (3), *dh* (3), *rand* (3), *RSA_new_method* (3)

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* (3) manpage describes how to access error codes.

Error codes contain information about where the error occurred, and what went wrong. *ERR_GET_LIB* (3) describes how to extract this information. A method to obtain human-readable error messages is described in *ERR_error_string* (3).

ERR_clear_error (3) can be used to clear the error queue.

Note that *ERR_remove_state* (3) should be used to avoid memory leaks when threads are terminated.

ADDING NEW ERROR CODES TO OPENSSL

See *ERR_put_error* (3)> if you want to record error codes in the OpenSSL error system from within your application.

The remainder of this section is of interest only if you want to add new error codes to OpenSSL or add error codes from external libraries.

Reporting errors

Each sub-library has a specific macro XXXerr() that is used to report errors. Its first argument is a function code B<XXX_F_...>, the second argument is a reason code B<XXX_R_...>. Function codes are derived from the function names; reason codes consist of textual error descriptions. For example, the function ssl23_read() 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 above example.

The trailing section of a reason code (after the "_R_") is translated into lower case and underscores changed to spaces.

When you are using new function or reason codes, run B<make errors>. The necessary B<define>s will then automatically be added to the sub-library'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 normally only done when a library wants to include ASN1 code which must use the ASN1err() macro.

Adding new libraries

When adding a new sub-library to OpenSSL, assign it a library number B<ERR_LIB_XXX>, define a macro XXXerr() (both in B<err.h>), add its name to B<ERR_str_libraries[(in B<crypto/err/err.c>), and add C<err_load_xxx_strings> to the ERR_load_crypto_strings() function (in B<crypto/err/err_all.c>). Finally, add an entry

```
LXXXxxx.hxxx_err.c
```

to B<crypto/err/openssl.ec>, and add B<xxx_err.c> to the Makefile. Running B<make errors> will then generate a file B<xxx_err.c>, and add all error codes used in the library to B<xxx.h>.

Additionally the library include file must have a certain form. Typically it will initially look like this:

```
#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 B<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 B<make errors> is run. The closing #endif etc will be automatically added by the script.

The generated C error code file B<xxx_err.c> will load the header files B<stdio.h>, B<openssl/err.h> and B<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 is also 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 B<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.

None. more details

INTERNALS

The error queues are stored in a hash table with one B<ERR_STATE> entry for each pid. ERR_get_state() returns the current thread's B<ERR_STATE>. An B<ERR_STATE> can hold up to B<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 table. The hash tables can be obtained by calling ERR_get_err_state_table(void) and ERR_get_string_table(void) respectively.

SEE ALSO

CRYPTO_set_locking_callback (3), *ERR_get_error* (3), *ERR_GET_LIB* (3), *ERR_clear_error* (3), *ERR_error_string* (3), *ERR_print_errors* (3), *ERR_load_crypto_strings* (3), *ERR_remove_state* (3), *ERR_put_error* (3), *ERR_load_strings* (3), *SSL_get_error* (3)

ERR_clear_error

NAME

ERR_clear_error – clear the error queue

Synopsis

```
#include <openssl/err.h>
void ERR_clear_error(void);
```

DESCRIPTION

ERR_clear_error() empties the current thread's error queue.

RETURN VALUES

ERR_clear_error() has no return value.

SEE ALSO

err (3), *ERR_get_error* (3)

HISTORY

ERR_clear_error() is available in all versions of SSLeay and OpenSSL.

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

ERR_error_string() generates a human-readable string representing the error code *e*, and places it at *buf*. *buf* must be at least 120 bytes long. If *buf* is *NULL*, the error string is placed in a static buffer. ERR_error_string_n() is a variant of ERR_error_string() that writes at most *len* characters (including the terminating 0) and truncates the string if necessary. For ERR_error_string_n(), *buf* may not be *NULL*.

The string will have the following format:

```
error:[error code]:[library name]:[function name]:[reason string]
```

error code is an 8 digit hexadecimal number, *library name*, *function name* and *reason string* are ASCII text.

ERR_lib_error_string(), ERR_func_error_string() and ERR_reason_error_string() return the library name, function name and reason string respectively.

The OpenSSL error strings should be loaded by calling *ERR_load_crypto_strings* (3) or, for SSL applications, *SSL_load_error_strings* (3) first. If there is no text string registered for the given error code, the error string will contain the numeric code.

ERR_print_errors (3) can be used to print all error codes currently in the queue.

RETURN VALUES

ERR_error_string() returns a pointer to a static buffer containing the string if *buf* == *NULL*, *buf* otherwise.

ERR_lib_error_string(), ERR_func_error_string() and ERR_reason_error_string() return the strings, and *NULL* if none is registered for the error code.

SEE ALSO

err (3), *ERR_get_error* (3), *ERR_load_crypto_strings* (3), *SSL_load_error_strings* (3) *ERR_print_errors* (3)

HISTORY

ERR_error_string() is available in all versions of SSLeay and OpenSSL. ERR_error_string_n() was added in OpenSSL 0.9.6.

ERR_get_error

NAME

ERR_get_error, ERR_peek_error, ERR_peek_last_error, ERR_get_error_line,

ERR_peek_error_line, ERR_peek_last_error_line, ERR_get_error_line_data,

ERR_peek_error_line_data, ERR_peek_last_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_peek_last_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_peek_last_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);
unsigned long ERR_peek_last_error_line_data(const char **file, int *line, const char
**data, int *flags);
```

DESCRIPTION

ERR_get_error() returns the earliest 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.

ERR_peek_error() returns the earliest error code from the thread's error queue without modifying it.

ERR_peek_last_error() returns the latest error code from the thread's error queue without modifying it.

See *ERR_GET_LIB* (3) for obtaining information about location and reason of the error, and *ERR_error_string* (3) for human-readable error messages.

ERR_get_error_line(), ERR_peek_error_line() and ERR_peek_last_error_line() are the same as the above, but they additionally store the file name and line number where the error occurred in **file* and **line*, unless these are *NULL*.

ERR_get_error_line_data(), ERR_peek_error_line_data() and ERR_get_last_error_line_data() store additional data and flags associated with the error code in **data* and **flags*, unless these are *NULL*. **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.

SEE ALSO

err (3), *ERR_error_string* (3), *ERR_GET_LIB* (3)

HISTORY

ERR_get_error(), ERR_peek_error(), ERR_get_error_line() and ERR_peek_error_line() are available in all versions of SSLeay and OpenSSL. ERR_get_error_line_data() and ERR_peek_error_line_data() were added in SSLeay 0.9.0. ERR_peek_last_error(), ERR_peek_last_error_line() and ERR_peek_last_error_line_data() were added in OpenSSL 0.9.7.

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 ERR_get_error() consists of a library number, function code and reason code. ERR_GET_LIB(), ERR_GET_FUNC() and ERR_GET_REASON() 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 sub-library of OpenSSL has a unique library number; function and reason codes are unique within each sub-library. Note that different libraries may use the same value to signal different functions and reasons.

ERR_R_... reason codes such as *ERR_R_MALLOC_FAILURE* are globally unique. However, when checking for sub-library specific reason codes, be sure to also compare the library number.

ERR_GET_LIB(), ERR_GET_FUNC() and ERR_GET_REASON() are macros.

RETURN VALUES

The library number, function code and reason code respectively.

SEE ALSO

err (3), *ERR_get_error* (3)

HISTORY

ERR_GET_LIB(), ERR_GET_FUNC() and ERR_GET_REASON() are available in all versions of SSLeay and OpenSSL.

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

ERR_load_crypto_strings() registers the error strings for all *libcrypto* functions. SSL_load_error_strings() 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.

ERR_free_strings() frees all previously loaded error strings.

RETURN VALUES

ERR_load_crypto_strings(), SSL_load_error_strings() and ERR_free_strings() return no values.

SEE ALSO

err (3), *ERR_error_string* (3)

HISTORY

ERR_load_error_strings(), SSL_load_error_strings() and ERR_free_strings() are available in all versions of SSLeay and OpenSSL.

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

ERR_load_strings() registers error strings for library number *lib*.

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}.

ERR_get_next_error_library() can be used to assign library numbers to user libraries at runtime.

RETURN VALUE

ERR_load_strings() returns no value. ERR_PACK() return the error code. ERR_get_next_error_library() returns a new library number.

SEE ALSO

err (3), *ERR_load_strings* (3)

HISTORY

ERR_load_error_strings() and ERR_PACK() are available in all versions of SSLeay and OpenSSL. ERR_get_next_error_library() was added in SSLeay 0.9.0.

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.

ERR_print_errors_fp() 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]
```

error code is an 8 digit hexadecimal number. *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

ERR_print_errors() and ERR_print_errors_fp() return no values.

SEE ALSO

err (3), *ERR_error_string* (3), *ERR_get_error* (3), *ERR_load_crypto_strings* (3), *SSL_load_error_strings* (3)

HISTORY

ERR_print_errors() and ERR_print_errors_fp() are available in all versions of SSLeay and OpenSSL.

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

ERR_put_error() 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.

ERR_add_error_data() associates the concatenation of its *num* string arguments with the error code added last.

ERR_load_strings (3) can be used to register error strings so that the application can a generate human-readable error messages for the error code.

RETURN VALUES

ERR_put_error() and ERR_add_error_data() return no values.

SEE ALSO

err (3), *ERR_load_strings* (3)

HISTORY

ERR_put_error() is available in all versions of SSLeay and OpenSSL. ERR_add_error_data() was added in SSLeay 0.9.0.

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

ERR_remove_state() 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

ERR_remove_state() returns no value.

SEE ALSO

err (3)

HISTORY

ERR_remove_state() is available in all versions of SSLeay and OpenSSL.

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* (3).

All the symmetric algorithms (ciphers) and digests can be replaced by ENGINE modules providing alternative implementations. If ENGINE implementations of ciphers or digests are registered as defaults, then the various EVP functions will automatically use those implementations automatically in preference to built in software implementations. For more information, consult the *engine* (3) man page.

SEE ALSO

EVP_DigestInit (3), *EVP_EncryptInit* (3), *EVP_OpenInit* (3), *EVP_SealInit* (3), *EVP_SignInit* (3), *EVP_VerifyInit* (3), *OpenSSL_add_all_algorithms* (3), *engine* (3)

EVP_BytesToKey

NAME

EVP_BytesToKey – password based encryption routine

Synopsis

```
#include <openssl/evp.h>
int EVP_BytesToKey(const EVP_CIPHER *type,const EVP_MD *md, const unsigned char *salt,
const unsigned char *data, int datal, int count, unsigned char *key,unsigned char *iv);
```

DESCRIPTION

EVP_BytesToKey() derives a key and IV from various parameters. *type* is the cipher to derive the key and IV for. *md* is the message digest to use. The *salt* paramter is used as a salt in the derivation: it should point to an 8 byte buffer or NULL if no salt is used. *data* is a buffer containing *datal* bytes which is used to derive the keying data. *count* is the iteration count to use. The derived key and IV will be written to *key* and *iv* respectively.

NOTES

A typical application of this function is to derive keying material for an encryption algorithm from a password in the *data* parameter.

Increasing the *count* parameter slows down the algorithm which makes it harder for an attacker to peform a brute force attack using a large number of candidate passwords.

If the total key and IV length is less than the digest length and *MD5* is used then the derivation algorithm is compatible with PKCS#5 v1.5 otherwise a non standard extension is used to derive the extra data.

Newer applications should use more standard algorithms such as PKCS#5 v2.0 for key derivation.

KEY DERIVATION ALGORITHM

The key and IV is derived by concatenating D_1, D_2, etc until enough data is available for the key and IV. D_i is defined as:

```
D_i = HASH^count(D_(i-1) || data || salt)
```

where || denotes concatentaion, D_0 is empty, HASH is the digest algorithm in use, HASH^1(data) is simply HASH(data), HASH^2(data) is HASH(HASH(data)) and so on.

The initial bytes are used for the key and the subsequent bytes for the IV.

RETURN VALUES

EVP_BytesToKey() returns the size of the derived key in bytes.

SEE ALSO

evp (3), *rand* (3), *EVP_EncryptInit* (3),

HISTORY

None.

EVP_MD_CTX_init

NAME

EVP_MD_CTX_init, EVP_MD_CTX_create, EVP_DigestInit_ex, EVP_DigestUpdate, EVP_DigestFinal_ex, EVP_MD_CTX_cleanup, EVP_MD_CTX_destroy, EVP_MAX_MD_SIZE, EVP_MD_CTX_copy_ex, 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_md5, EVP_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_MD_CTX_init(EVP_MD_CTX *ctx);
VP_MD_CTX *EVP_MD_CTX_create(void);
int EVP_DigestInit_ex(EVP_MD_CTX *ctx, const EVP_MD *type, ENGINE *impl);
int EVP_DigestUpdate(EVP_MD_CTX *ctx, const void *d, unsigned int cnt);
int EVP_DigestFinal_ex(EVP_MD_CTX *ctx, unsigned char *md, unsigned int *s);
int EVP_MD_CTX_cleanup(EVP_MD_CTX *ctx);
void EVP_MD_CTX_destroy(EVP_MD_CTX *ctx);
int EVP_MD_CTX_copy_ex(EVP_MD_CTX *out,const EVP_MD_CTX *in);
int EVP_DigestInit(EVP_MD_CTX *ctx, const EVP_MD *type);
int EVP_DigestFinal(EVP_MD_CTX *ctx, unsigned char *md, unsigned int *s);
int EVP_MD_CTX_copy(EVP_MD_CTX *out,EVP_MD_CTX *in);
#define EVP_MAX_MD_SIZE (16+20) /* The SSLv3 md5+sha1 type */
#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) const EVP_MD *EVP_md_null(void);
const EVP_MD *EVP_md2(void); const EVP_MD *EVP_md5(void);
const EVP_MD *EVP_sha(void); const EVP_MD *EVP_sha1(void);
const EVP_MD *EVP_dss(void); const EVP_MD *EVP_dss1(void);
const EVP_MD *EVP_mdc2(void); const 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.

EVP_MD_CTX_init() initializes digest contet *ctx*.

EVP_MD_CTX_create() allocates, initializes and returns a digest contet.

EVP_DigestInit_ex() sets up digest context *ctx* to use a digest *type* from ENGINE *impl*. *ctx* must be initialized before calling this function. *type* will typically be supplied by a functionsuch as EVP_sha1(). If *impl* is NULL then the default implementation of digest *type* is used.

EVP_DigestUpdate() 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.

EVP_DigestFinal_ex() 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_ex() no additional calls to EVP_DigestUpdate() can be made, but EVP_DigestInit_ex() can be called to initialize a new digest operation.

EVP_MD_CTX_cleanup() cleans up digest context *ctx*, it should be called after a digest context is no longer needed.

EVP_MD_CTX_destroy() cleans up digest context *ctx* and frees up the space allocated to it, it should be called only on a context created using EVP_MD_CTX_create().

EVP_MD_CTX_copy_ex() can be used to copy the message digest state from *in* to *out*. This is useful if large amounts of data are to be hashed which only differ in the last few bytes. *out* must be initialized before calling this function.

EVP_DigestInit() behaves in the same way as EVP_DigestInit_ex() except the passed context *ctx* does not have to be initialized, and it always uses the default digest implementation.

EVP_DigestFinal() is similar to EVP_DigestFinal_ex() except the digest contet *ctx* is automatically cleaned up.

EVP_MD_CTX_copy() is similar to EVP_MD_CTX_copy_ex() except the destination *out* does not have to be initialized.

EVP_MD_size() and EVP_MD_CTX_size() 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.

EVP_MD_block_size() and EVP_MD_CTX_block_size() return the block size of the message digest when passed an *EVP_MD* or an *EVP_MD_CTX* structure.

EVP_MD_type() and EVP_MD_CTX_type() 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.

EVP_MD_CTX_md() returns the *EVP_MD* structure corresponding to the passed *EVP_MD_CTX*.

EVP_MD_pkey_type() 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*. This "link" between digests and signature algorithms may not be retained in future versions of OpenSSL.

EVP_md2(), EVP_md5(), EVP_sha(), EVP_sha1(), EVP_mdc2() and EVP_ripemd160() 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.

EVP_dss() and EVP_dss1() return *EVP_MD* structures for SHA and SHA1 digest algorithms but using DSS (DSA) for the signature algorithm.

EVP_md_null() is a "null" message digest that does nothing: i.e. the hash it returns is of zero length.

EVP_get_digestbyname(), EVP_get_digestbynid() and EVP_get_digestbyobj() 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.

RETURN VALUES

EVP_DigestInit_ex(), EVP_DigestUpdate() and EVP_DigestFinal_ex() return 1 for success and 0 for failure.

EVP_MD_CTX_copy_ex() returns 1 if successful or 0 for failure.

EVP_MD_type(), EVP_MD_pkey_type() and EVP_MD_type() return the NID of the corresponding OBJECT IDENTIFIER or NID_undef if none exists.

EVP_MD_size(), EVP_MD_block_size(), EVP_MD_CTX_size(e), EVP_MD_size(), EVP_MD_CTX_block_size() and EVP_MD_block_size() return the digest or block size in bytes.

EVP_md_null(), EVP_md2(), EVP_md5(), EVP_sha(), EVP_sha1(), EVP_dss(), EVP_dss1(), EVP_mdc2() and EVP_ripemd160() return pointers to the corresponding EVP_MD structures.

EVP_get_digestbyname(), EVP_get_digestbynid() and EVP_get_digestbyobj() return either an *EVP_MD* structure or NULL if an error occurs.

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.

For most applications the *impl* parameter to EVP_DigestInit_ex() will be set to NULL to use the default digest implementation.

The functions EVP_DigestInit(), EVP_DigestFinal() and EVP_MD_CTX_copy() are obsolete but are retained to maintain compatibility with existing code. New applications should use EVP_DigestInit_ex(), EVP_DigestFinal_ex() and EVP_MD_CTX_copy_ex() because they can efficiently reuse a digest context instead of initializing and cleaning it up on each call and allow non default implementations of digests to be specified.

In OpenSSL 0.9.7 and later if digest contexts are not cleaned up after use memory leaks will occur.

EXAMPLE

This 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_MD_CTX_init(&mdctx);
 EVP_DigestInit_ex(&mdctx, md, NULL);
 EVP_DigestUpdate(&mdctx, mess1, strlen(mess1));
 EVP_DigestUpdate(&mdctx, mess2, strlen(mess2));
 EVP_DigestFinal_ex(&mdctx, md_value, &md_len);
 EVP_MD_CTX_cleanup(&mdctx);

 printf("Digest is: ");
 for(i = 0; i < md_len; i++) printf("%02x", md_value[i]);
 printf("\n");
 }
```

Restrictions

The link between digests and signing algorithms results in a situation where EVP_sha1() must be used with RSA and EVP_dss1() must be used with DSS even though they are identical digests.

SEE ALSO

evp (3), *hmac* (3), `md2(3)`, `md5(3)`, `mdc2(3)`, *ripemd* (3), *sha* (3), *dgst* (1)

HISTORY

EVP_DigestInit(), EVP_DigestUpdate() and EVP_DigestFinal() are available in all versions of SSLeay and OpenSSL.

EVP_MD_CTX_init(), EVP_MD_CTX_create(), EVP_MD_CTX_copy_ex(), EVP_MD_CTX_cleanup(), EVP_MD_CTX_destroy(), EVP_DigestInit_ex() and EVP_DigestFinal_ex() were added in OpenSSL 0.9.7.

EVP_md_null(), EVP_md2(), EVP_md5(), EVP_sha(), EVP_sha1(), EVP_dss(), EVP_dss1(), EVP_mdc2() and EVP_ripemd160() were changed to return truely const EVP_MD * in OpenSSL 0.9.7.

EVP_CIPHER_CTX_init

NAME

EVP_CIPHER_CTX_init, EVP_EncryptInit_ex, EVP_EncryptUpdate, EVP_EncryptFinal_ex, EVP_DecryptInit_ex, EVP_DecryptUpdate, EVP_DecryptFinal_ex, EVP_CipherInit_ex, EVP_CipherUpdate, EVP_CipherFinal_ex, EVP_CIPHER_CTX_set_key_length, EVP_CIPHER_CTX_ctrl, EVP_CIPHER_CTX_cleanup, EVP_EncryptInit, EVP_EncryptFinal, EVP_DecryptInit, EVP_DecryptFinal, EVP_CipherInit, EVP_CipherFinal, EVP_get_cipherbyname, 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_CTX_set_padding – EVP cipher routines

Synopsis

```
#include <openssl/evp.h>
int EVP_CIPHER_CTX_init(EVP_CIPHER_CTX *a);
int EVP_EncryptInit_ex(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type, ENGINE *impl, 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_ex(EVP_CIPHER_CTX *ctx, unsigned char *out, int *outl);
int EVP_DecryptInit_ex(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type, ENGINE *impl, 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_ex(EVP_CIPHER_CTX *ctx, unsigned char *outm, int *outl);
int EVP_CipherInit_ex(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type, ENGINE *impl, 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_ex(EVP_CIPHER_CTX *ctx, unsigned char *outm, int *outl);
int EVP_EncryptInit(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type, unsigned char *key,
unsigned char *iv);
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_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_CipherFinal(EVP_CIPHER_CTX *ctx, unsigned char *outm, int *outl);
int EVP_CIPHER_CTX_set_padding(EVP_CIPHER_CTX *x, int padding);
```

```
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.

EVP_CIPHER_CTX_init() initializes cipher contex *ctx*.

EVP_EncryptInit_ex() sets up cipher context *ctx* for encryption with cipher *type* from ENGINE *impl*. *ctx* must be initialized before calling this function. *type* is normally supplied by a function such as EVP_des_cbc(). If *impl* is NULL then the default implementation is used. *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.

EVP_EncryptUpdate() 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*.

If padding is enabled (the default) then EVP_EncryptFinal_ex() encrypts the "final" data, that is any data that remains in a partial block. It uses `standard block padding` (aka 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.

If padding is disabled then EVP_EncryptFinal_ex() will not encrypt any more data and it will return an error if any data remains in a partial block: that is if the total data length is not a multiple of the block size.

EVP_DecryptInit_ex(), EVP_DecryptUpdate() and EVP_DecryptFinal_ex() are the corresponding decryption operations. EVP_DecryptFinal() will return an error code if padding is enabled and the final block is not correctly formatted. The parameters and restrictions are identical to the encryption operations except that if padding is enabled 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.

EVP_CipherInit_ex(), EVP_CipherUpdate() and EVP_CipherFinal_ex() are functions that 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).

EVP_CIPHER_CTX_cleanup() clears all information from a cipher context and free up any allocated memory associate with it. It should be called after all operations using a cipher are complete so sensitive information does not remain in memory.

EVP_EncryptInit(), EVP_DecryptInit() and EVP_CipherInit() behave in a similar way to EVP_EncryptInit_ex(), EVP_DecryptInit_ex and EVP_CipherInit_ex() except the *ctx* paramter does not need to be initialized and they always use the default cipher implementation.

EVP_EncryptFinal(), EVP_DecryptFinal() and EVP_CipherFinal() behave in a similar way to EVP_EncryptFinal_ex(), EVP_DecryptFinal_ex() and EVP_CipherFinal_ex() except *ctx* is automatically cleaned up after the call.

EVP_get_cipherbyname(), EVP_get_cipherbynid() and EVP_get_cipherbyobj() return an EVP_CIPHER structure when passed a cipher name, a NID or an ASN1_OBJECT structure.

EVP_CIPHER_nid() and EVP_CIPHER_CTX_nid() 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.

EVP_CIPHER_CTX_set_padding() enables or disables padding. By default encryption operations are padded using standard block padding and the padding is checked and removed when decrypting. If the *pad* parameter is zero then no padding is performed, the total amount of data encrypted or decrypted must then be a multiple of the block size or an error will occur.

EVP_CIPHER_key_length() and EVP_CIPHER_CTX_key_length() 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. Note: although EVP_CIPHER_key_length() is fixed for a given cipher, the value of EVP_CIPHER_CTX_key_length() may be different for variable key length ciphers.

EVP_CIPHER_CTX_set_key_length() 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.

EVP_CIPHER_iv_length() and EVP_CIPHER_CTX_iv_length() 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.

EVP_CIPHER_block_size() and EVP_CIPHER_CTX_block_size() 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.

EVP_CIPHER_type() and EVP_CIPHER_CTX_type() 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*.

EVP_CIPHER_CTX_cipher() returns the *EVP_CIPHER* structure when passed an *EVP_CIPHER_CTX* structure.

EVP_CIPHER_mode() and EVP_CIPHER_CTX_mode() 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.

EVP_CIPHER_param_to_asn1() sets the AlgorithmIdentifier "parameter" based on the passed cipher. This will typically 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(), EVP_DecryptUpdate() calls for example). This function may fail if the cipher does not have any ASN1 support.

EVP_CIPHER_asn1_to_param() 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 EVP_CipherInit() will be called with the IV and key set to NULL, EVP_CIPHER_asn1_to_param() will be called and finally EVP_CipherInit() again with all parameters except the key 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.

EVP_CIPHER_CTX_ctrl() 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.

RETURN VALUES

EVP_CIPHER_CTX_init, EVP_EncryptInit_ex(), EVP_EncryptUpdate() and EVP_EncryptFinal_ex() return 1 for success and 0 for failure.

EVP_DecryptInit_ex() and EVP_DecryptUpdate() return 1 for success and 0 for failure. EVP_DecryptFinal_ex() returns 0 if the decrypt failed or 1 for success.

EVP_CipherInit_ex() and EVP_CipherUpdate() return 1 for success and 0 for failure. EVP_CipherFinal_ex() returns 0 for a decryption failure or 1 for success.

EVP_CIPHER_CTX_cleanup() returns 1 for success and 0 for failure.

EVP_get_cipherbyname(), EVP_get_cipherbynid() and EVP_get_cipherbyobj() return an *EVP_CIPHER* structure or NULL on error.

EVP_CIPHER_nid() and EVP_CIPHER_CTX_nid() return a NID.

EVP_CIPHER_block_size() and EVP_CIPHER_CTX_block_size() return the block size.

EVP_CIPHER_key_length() and EVP_CIPHER_CTX_key_length() return the key length.

EVP_CIPHER_CTX_set_padding() always returns 1.

EVP_CIPHER_iv_length() and EVP_CIPHER_CTX_iv_length() return the IV length or zero if the cipher does not use an IV.

EVP_CIPHER_type() and EVP_CIPHER_CTX_type() return the NID of the cipher's OBJECT IDENTIFIER or NID_undef if it has no defined OBJECT IDENTIFIER.

EVP_CIPHER_CTX_cipher() returns an *EVP_CIPHER* structure.

EVP_CIPHER_param_to_asn1() and EVP_CIPHER_asn1_to_param() return 1 for success or zero for failure.

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 EVP_rc4() and the EVP_CIPHER_CTX_set_key_length() function.

- 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 EVP_rc2_cbc(), EVP_CIPHER_CTX_set_key_length() and EVP_CIPHER_CTX_ctrl() 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 if padding is enabled, it is not a strong test that the input data or key is correct. A random block has better than 1 in 256 chance of being of the correct format and problems with the input data earlier on will not produce a final decrypt error.

If padding is disabled then the decryption operation will always succeed if the total amount of data decrypted is a multiple of the block size.

The functions EVP_EncryptInit(), EVP_EncryptFinal(), EVP_DecryptInit(), EVP_CipherInit() and EVP_CipherFinal() are obsolete but are retained for compatibility with existing code. New code should use EVP_EncryptInit_ex(), EVP_EncryptFinal_ex(), EVP_DecryptInit_ex(), EVP_DecryptFinal_ex(), EVP_CipherInit_ex() and EVP_CipherFinal_ex() because they can reuse an existing context without allocating and freeing it up on each call.

Restrictions

For RC5 the number of rounds can currently only be set to 8, 12 or 16. This is a limitation of the current RC5 code rather than the EVP interface.

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.

EXAMPLES

Get the number of rounds used in RC5:

```
int nrounds;
EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GET_RC5_ROUNDS, 0, &nrounds);
```

Get the RC2 effective key length:

```
int key_bits;
EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GET_RC2_KEY_BITS, 0, &key_bits);
```

Set the number of rounds used in RC5:

```
int nrounds;
EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_SET_RC5_ROUNDS, nrounds, NULL);
```

Set the effective key length used in RC2:

```
int key_bits;
EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_SET_RC2_KEY_BITS, key_bits, NULL);
```

Encrypt a string using blowfish:

```
int do_crypt(char *outfile)
{
unsigned char outbuf[1024];
int outlen, tmplen;
```

```
/* Bogus key and IV: we'd normally set these from
 * another source.
 */
unsigned char key[] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
unsigned char iv[] = {1,2,3,4,5,6,7,8};
char intext[] = "Some Crypto Text";
EVP_CIPHER_CTX ctx;
FILE *out;
EVP_CIPHER_CTX_init(&ctx);
EVP_EncryptInit_ex(&ctx, EVP_bf_cbc(), NULL, key, iv);

if(!EVP_EncryptUpdate(&ctx, outbuf, &outlen, intext, strlen(intext)))
{
/* Error */
return 0;
}
/* Buffer passed to EVP_EncryptFinal() must be after data just
 * encrypted to avoid overwriting it.
 */
if(!EVP_EncryptFinal_ex(&ctx, outbuf + outlen, &tmplen))
{
/* Error */
return 0;
}
outlen += tmplen;
EVP_CIPHER_CTX_cleanup(&ctx);
/* Need binary mode for fopen because encrypted data is
 * binary data. Also cannot use strlen() on it because
         * it wont be null terminated and may contain embedded
 * nulls.
 */
out = fopen(outfile, "wb");
fwrite(outbuf, 1, outlen, out);
fclose(out);
return 1;
}
```

The ciphertext from the above example can be decrypted using the *openssl* utility with the command line:

```
S<openssl bf -in cipher.bin -K 000102030405060708090A0B0C0D0E0F -iv 0102030405060708 -d>
```

General encryption, decryption function example using FILE I/O and RC2 with an 80 bit key:

```
 int do_crypt(FILE *in, FILE *out, int do_encrypt)
 {
/* Allow enough space in output buffer for additional block */
inbuf[1024], outbuf[1024 + EVP_MAX_BLOCK_LENGTH];
int inlen, outlen;
/* Bogus key and IV: we'd normally set these from
 * another source.
 */
unsigned char key[] = "0123456789";
unsigned char iv[] = "12345678";
/* Don't set key or IV because we will modify the parameters */
EVP_CIPHER_CTX_init(&ctx);
EVP_CipherInit_ex(&ctx, EVP_rc2(), NULL, NULL, NULL, do_encrypt);
EVP_CIPHER_CTX_set_key_length(&ctx, 10);
/* We finished modifying parameters so now we can set key and IV */
EVP_CipherInit_ex(&ctx, NULL, NULL, key, iv, do_encrypt);
```

```
for(;;)
{
inlen = fread(inbuf, 1, 1024, in);
if(inlen <= 0) break;
if(!EVP_CipherUpdate(&ctx, outbuf, &outlen, inbuf, inlen))
{
/* Error */
return 0;
}
fwrite(outbuf, 1, outlen, out);
}
if(!EVP_CipherFinal_ex(&ctx, outbuf, &outlen))
{
/* Error */
return 0;
}
fwrite(outbuf, 1, outlen, out);

EVP_CIPHER_CTX_cleanup(&ctx);
return 1;
}
```

SEE ALSO

evp (3)

HISTORY

EVP_CIPHER_CTX_init(), EVP_EncryptInit_ex(), EVP_EncryptFinal_ex(), EVP_DecryptInit_ex(), EVP_DecryptFinal_ex(), EVP_CipherInit_ex(), EVP_CipherFinal_ex() and EVP_CIPHER_CTX_set_padding() appeared in OpenSSL 0.9.7.

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.

EVP_OpenInit() 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.

EVP_OpenUpdate() and EVP_OpenFinal() have exactly the same properties as the EVP_DecryptUpdate() and EVP_DecryptFinal() routines, as documented on the *EVP_EncryptInit* (3) manual 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

EVP_OpenInit() returns 0 on error or a non zero integer (actually the recovered secret key size) if successful.

EVP_OpenUpdate() returns 1 for success or 0 for failure.

EVP_OpenFinal() returns 0 if the decrypt failed or 1 for success.

SEE ALSO

evp (3), *rand* (3), *EVP_EncryptInit* (3), *EVP_SealInit* (3)

HISTORY

None.

EVP_PKEY_new

NAME

EVP_PKEY_new, EVP_PKEY_free – private key allocation functions.

Synopsis

```
#include <openssl/evp.h>
EVP_PKEY *EVP_PKEY_new(void);
void EVP_PKEY_free(EVP_PKEY *key);
```

DESCRIPTION

The EVP_PKEY_new() function allocates an empty *EVP_PKEY* structure which is used by OpenSSL to store private keys.

EVP_PKEY_free() frees up the private key *key*.

NOTES

The *EVP_PKEY* structure is used by various OpenSSL functions which require a general private key without reference to any particular algorithm.

The structure returned by EVP_PKEY_new() is empty. To add a private key to this empty structure the functions described in EVP_PKEY_set1_*RSA* (3) should be used.

RETURN VALUES

EVP_PKEY_new() returns either the newly allocated *EVP_PKEY* structure of *NULL* if an error occurred.

EVP_PKEY_free() does not return a value.

SEE ALSO

EVP_PKEY_set1_*RSA* (3)

HISTORY

None.

EVP_PKEY_set1_RSA

NAME

EVP_PKEY_set1_RSA, EVP_PKEY_set1_DSA, EVP_PKEY_set1_DH, EVP_PKEY_set1_EC_KEY, EVP_PKEY_get1_RSA, EVP_PKEY_get1_DSA, EVP_PKEY_get1_DH, EVP_PKEY_get1_EC_KEY, EVP_PKEY_assign_RSA, EVP_PKEY_assign_DSA, EVP_PKEY_assign_DH, EVP_PKEY_assign_EC_KEY, EVP_PKEY_type – EVP_PKEY assignment functions.

Synopsis

```
#include <openssl/evp.h>
int EVP_PKEY_set1_RSA(EVP_PKEY *pkey,RSA *key);
int EVP_PKEY_set1_DSA(EVP_PKEY *pkey,DSA *key);
int EVP_PKEY_set1_DH(EVP_PKEY *pkey,DH *key);
int EVP_PKEY_set1_EC_KEY(EVP_PKEY *pkey,EC_KEY *key);
RSA *EVP_PKEY_get1_RSA(EVP_PKEY *pkey);
DSA *EVP_PKEY_get1_DSA(EVP_PKEY *pkey);
DH *EVP_PKEY_get1_DH(EVP_PKEY *pkey);
EC_KEY *EVP_PKEY_get1_EC_KEY(EVP_PKEY *pkey);
int EVP_PKEY_assign_RSA(EVP_PKEY *pkey,RSA *key);
int EVP_PKEY_assign_DSA(EVP_PKEY *pkey,DSA *key);
int EVP_PKEY_assign_DH(EVP_PKEY *pkey,DH *key);
int EVP_PKEY_assign_EC_KEY(EVP_PKEY *pkey,EC_KEY *key);
int EVP_PKEY_type(int type);
```

DESCRIPTION

EVP_PKEY_set1_RSA(), EVP_PKEY_set1_DSA(), EVP_PKEY_set1_DH() and EVP_PKEY_set1_EC_KEY() set the key referenced by *pkey* to *key*.

EVP_PKEY_get1_RSA(), EVP_PKEY_get1_DSA(), EVP_PKEY_get1_DH() and EVP_PKEY_get1_EC_KEY() return the referenced key in *pkey* or *NULL* if the key is not of the correct type.

EVP_PKEY_assign_RSA() EVP_PKEY_assign_DSA(), EVP_PKEY_assign_DH() and EVP_PKEY_assign_EC_KEY() also set the referenced key to *key* however these use the supplied *key* internally and so *key* will be freed when the parent *pkey* is freed.

EVP_PKEY_type() returns the type of key corresponding to the value *type*. The type of a key can be obtained with EVP_PKEY_type(pkey->type). The return value will be EVP_PKEY_RSA, EVP_PKEY_DSA, EVP_PKEY_DH or EVP_PKEY_EC for the corresponding key types or NID_undef if the key type is unassigned.

NOTES

In accordance with the OpenSSL naming convention the key obtained from or assigned to the *pkey* using the *1* functions must be freed as well as *pkey*.

EVP_PKEY_assign_RSA() EVP_PKEY_assign_DSA(), EVP_PKEY_assign_DH() EVP_PKEY_assign_EC_KEY() are implemented as macros.

RETURN VALUES

EVP_PKEY_set1_RSA(), EVP_PKEY_set1_DSA(), EVP_PKEY_set1_DH() and EVP_PKEY_set1_EC_KEY() return 1 for success or 0 for failure.

EVP_PKEY_get1_RSA(), EVP_PKEY_get1_DSA(), EVP_PKEY_get1_DH() and EVP_PKEY_get1_EC_KEY() return the referenced key or *NULL* if an error occurred.

EVP_PKEY_assign_RSA() EVP_PKEY_assign_DSA(), EVP_PKEY_assign_DH() and EVP_PKEY_assign_EC_KEY() return 1 for success and 0 for failure.

SEE ALSO

EVP_PKEY_new (3)

HISTORY

None.

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 IV (if required) then "envelope" it by using public key encryption. Data can then be encrypted using this key.

EVP_SealInit() initializes a cipher context *ctx* for encryption with cipher *type* using a random secret key and IV. *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. *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*. *pubk* is an array of *npubk* public keys.

The *iv* parameter is a buffer where the generated IV is written to. It must contain enough room for the corresponding cipher's IV, as determined by (for example) EVP_CIPHER_iv_length(type).

If the cipher does not require an IV then the *iv* parameter is ignored and can be *NULL*.

EVP_SealUpdate() and EVP_SealFinal() have exactly the same properties as the EVP_EncryptUpdate() and EVP_EncryptFinal() routines, as documented on the *EVP_EncryptInit* (3) manual page.

RETURN VALUES

EVP_SealInit() returns 0 on error or *npubk* if successful.

EVP_SealUpdate() and EVP_SealFinal() return 1 for success and 0 for failure.

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.

SEE ALSO

evp (3), *rand* (3), *EVP_EncryptInit* (3), *EVP_OpenInit* (3)

HISTORY

EVP_SealFinal() did not return a value before OpenSSL 0.9.7.

EVP_SignInit

NAME

EVP_SignInit, EVP_SignUpdate, EVP_SignFinal – EVP signing functions

Synopsis

```
#include <openssl/evp.h>
int EVP_SignInit_ex(EVP_MD_CTX *ctx, const EVP_MD *type, ENGINE *impl);
int 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);
void EVP_SignInit(EVP_MD_CTX *ctx, const EVP_MD *type);
int EVP_PKEY_size(EVP_PKEY *pkey);
```

DESCRIPTION

The EVP signature routines are a high level interface to digital signatures.

EVP_SignInit_ex() sets up signing context *ctx* to use digest *type* from ENGINE *impl*. *ctx* must be initialized with EVP_MD_CTX_init() before calling this function.

EVP_SignUpdate() 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.

EVP_SignFinal() 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.

EVP_SignInit() initializes a signing context *ctx* to use the default implementation of digest *type*.

EVP_PKEY_size() returns the maximum size of a signature in bytes. The actual signature returned by EVP_SignFinal() may be smaller.

RETURN VALUES

EVP_SignInit_ex(), EVP_SignUpdate() and EVP_SignFinal() return 1 for success and 0 for failure.

EVP_PKEY_size() returns the maximum size of a signature in bytes.

The error codes can be obtained by *ERR_get_error* (3).

NOTES

The *EVP* interface to digital signatures should almost always 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* (3).

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.

The call to EVP_SignFinal() internally finalizes a copy of the digest context. This means that calls to EVP_SignUpdate() and EVP_SignFinal() can be called later to digest and sign additional data.

Since only a copy of the digest context is ever finalized the context must be cleaned up after use by calling EVP_MD_CTX_cleanup() or a memory leak will occur.

Restrictions

Older versions of this documentation wrongly stated that calls to EVP_SignUpdate() could not be made after calling EVP_SignFinal().

SEE ALSO

EVP_VerifyInit (3), *EVP_DigestInit* (3), *err* (3), *evp* (3), *hmac* (3), `md2(3)`, `md5(3)`, `mdc2(3)`, *ripemd* (3), *sha* (3), *dgst* (1)

HISTORY

EVP_SignInit(), EVP_SignUpdate() and EVP_SignFinal() are available in all versions of SSLeay and OpenSSL.

EVP_SignInit_ex() was added in OpenSSL 0.9.7.

EVP_VerifyInit

NAME

EVP_VerifyInit, EVP_VerifyUpdate, EVP_VerifyFinal – EVP signature verification functions

Synopsis

```
#include <openssl/evp.h>
int EVP_VerifyInit_ex(EVP_MD_CTX *ctx, const EVP_MD *type, ENGINE *impl);
int 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);
int EVP_VerifyInit(EVP_MD_CTX *ctx, const EVP_MD *type);
```

DESCRIPTION

The EVP signature verification routines are a high level interface to digital signatures.

EVP_VerifyInit_ex() sets up verification context *ctx* to use digest *type* from ENGINE *impl*. *ctx* must be initialized by calling EVP_MD_CTX_init() before calling this function.

EVP_VerifyUpdate() 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.

EVP_VerifyFinal() verifies the data in *ctx* using the public key *pkey* and against the *siglen* bytes at *sigbuf*.

EVP_VerifyInit() initializes verification context *ctx* to use the default implementation of digest *type*.

RETURN VALUES

EVP_VerifyInit_ex() and EVP_VerifyUpdate() return 1 for success and 0 for failure.

EVP_VerifyFinal() returns 1 for a correct signature, 0 for failure and -1 if some other error occurred.

The error codes can be obtained by *ERR_get_error* (3).

NOTES

The *EVP* interface to digital signatures should almost always 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* (3).

The call to EVP_VerifyFinal() internally finalizes a copy of the digest context. This means that calls to EVP_VerifyUpdate() and EVP_VerifyFinal() can be called later to digest and verify additional data.

Since only a copy of the digest context is ever finalized the context must be cleaned up after use by calling EVP_MD_CTX_cleanup() or a memory leak will occur.

Restrictions

Older versions of this documentation wrongly stated that calls to EVP_VerifyUpdate() could not be made after calling EVP_VerifyFinal().

SEE ALSO

evp (3), *EVP_SignInit* (3), *EVP_DigestInit* (3), *err* (3), *evp* (3), *hmac* (3), `md2(3)`, `md5(3)`, `mdc2(3)`, *ripemd* (3), *sha* (3), *dgst* (1)

HISTORY

EVP_VerifyInit(), EVP_VerifyUpdate() and EVP_VerifyFinal() are available in all versions of SSLeay and OpenSSL.

EVP_VerifyInit_ex() was added in OpenSSL 0.9.7

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_CTX_init(HMAC_CTX *ctx);
void HMAC_Init(HMAC_CTX *ctx, const void *key, int key_len, const EVP_MD *md);
void HMAC_Init_ex(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_CTX_cleanup(HMAC_CTX *ctx);
void HMAC_cleanup(HMAC_CTX *ctx);
```

DESCRIPTION

HMAC is a MAC (message authentication code), i.e. a keyed hash function used for message authentication, which is based on a hash function.

HMAC() 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*.

evp_md can be EVP_sha1(), EVP_ripemd160() etc. *key* and *evp_md* may be *NULL* if a key and hash function have been set in a previous call to HMAC_Init() for that *HMAC_CTX*.

HMAC_CTX_init() initialises a *HMAC_CTX* before first use. It must be called.

HMAC_CTX_cleanup() erases the key and other data from the *HMAC_CTX* and releases any associated resources. It must be called when an *HMAC_CTX* is no longer required.

HMAC_cleanup() is an alias for HMAC_CTX_cleanup() included for back compatibility with 0.9.6b, it is deprecated.

The following functions may be used if the message is not completely stored in memory:

HMAC_Init() initializes a *HMAC_CTX* structure to use the hash function *evp_md* and the key *key* which is *key_len* bytes long. It is deprecated and only included for backward compatibility with OpenSSL 0.9.6b.

HMAC_Init_ex() initializes or reuses a *HMAC_CTX* structure to use the function *evp_md* and key *key*. Either can be NULL, in which case the existing one will be reused. HMAC_CTX_init() must have been called before the first use of an *HMAC_CTX* in this function. *N.B. HMAC_Init() had this undocumented behaviour in previous versions of OpenSSL - failure to switch to HMAC_Init_ex() in programs that expect it will cause them to stop working*.

HMAC_Update() can be called repeatedly with chunks of the message to be authenticated (*len* bytes at *data*).

HMAC_Final() places the message authentication code in *md*, which must have space for the hash function output.

RETURN VALUES

HMAC() returns a pointer to the message authentication code.

HMAC_CTX_init(), HMAC_Init_ex(), HMAC_Update(), HMAC_Final() and HMAC_CTX_cleanup() do not return values.

CONFORMING TO

RFC 2104

SEE ALSO

sha (3), *evp* (3)

HISTORY

HMAC(), HMAC_Init(), HMAC_Update(), HMAC_Final() and HMAC_cleanup() are available since SSLeay 0.9.0.

HMAC_CTX_init(), HMAC_Init_ex() and HMAC_CTX_cleanup() are available since OpenSSL 0.9.7.

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. This is mostly a legacy of Eric Young writing this library for the reasons of implementing what looked like a nice algorithm rather than for a particular software product.

lh_stats() 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.

lh_node_stats() prints the number of entries for each 'bucket' in the hash table.

lh_node_usage_stats() 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.

lh_stats_bio(), lh_node_stats_bio() and lh_node_usage_stats_bio() are the same as the above, except that the output goes to a *BIO*.

RETURN VALUES

These functions do not return values.

SEE ALSO

bio (3), *lhash* (3)

HISTORY

These functions are available in all versions of SSLeay and OpenSSL.

This manpage is derived from the SSLeay documentation.

lh_new

NAME

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(LHASH_HASH_FN_TYPE hash, LHASH_COMP_FN_TYPE compare);
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, LHASH_DOALL_FN_TYPE func);
void lh_doall_arg(LHASH *table, LHASH_DOALL_ARG_FN_TYPE func, void *arg);
int lh_error(LHASH *table);
typedef int (*LHASH_COMP_FN_TYPE)(const void *, const void *);
typedef unsigned long (*LHASH_HASH_FN_TYPE)(const void *);
typedef void (*LHASH_DOALL_FN_TYPE)(const void *);
typedef void (*LHASH_DOALL_ARG_FN_TYPE)(const void *, const void *);
```

DESCRIPTION

This library implements dynamic hash tables. The hash table entries can be arbitrary structures. Usually they consist of key and value fields.

lh_new() creates a new *LHASH* structure to store arbitrary data entries, and provides the 'hash' and 'compare' callbacks to be used in organising the table's entries. The *hash* callback takes a pointer to a table entry as its argument and returns an unsigned long hash value for 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* callback takes two arguments (pointers to two hash table entries), and returns 0 if their keys are equal, non-zero otherwise. If your hash table will contain items of some particular type and the *hash* and *compare* callbacks hash/compare these types, then the *DECLARE_LHASH_HASH_FN* and *IMPLEMENT_LHASH_COMP_FN* macros can be used to create callback wrappers of the prototypes required by lh_new(). These provide per-variable casts before calling the type-specific callbacks written by the application author. These macros, as well as those used for the "doall" callbacks, are defined as;

```
#define DECLARE_LHASH_HASH_FN(f_name,o_type) \
        unsigned long f_name##_LHASH_HASH(const void *);
#define IMPLEMENT_LHASH_HASH_FN(f_name,o_type) \
        unsigned long f_name##_LHASH_HASH(const void *arg) { \
                o_type a = (o_type)arg; \
                return f_name(a); }
#define LHASH_HASH_FN(f_name) f_name##_LHASH_HASH

#define DECLARE_LHASH_COMP_FN(f_name,o_type) \
        int f_name##_LHASH_COMP(const void *, const void *);
#define IMPLEMENT_LHASH_COMP_FN(f_name,o_type) \
        int f_name##_LHASH_COMP(const void *arg1, const void *arg2) { \
                o_type a = (o_type)arg1; \
                o_type b = (o_type)arg2; \
                return f_name(a,b); }
#define LHASH_COMP_FN(f_name) f_name##_LHASH_COMP
```

```
#define DECLARE_LHASH_DOALL_FN(f_name,o_type) \
        void f_name##_LHASH_DOALL(const void *);
#define IMPLEMENT_LHASH_DOALL_FN(f_name,o_type) \
        void f_name##_LHASH_DOALL(const void *arg) { \
                o_type a = (o_type)arg; \
                f_name(a); }
#define LHASH_DOALL_FN(f_name) f_name##_LHASH_DOALL

#define DECLARE_LHASH_DOALL_ARG_FN(f_name,o_type,a_type) \
        void f_name##_LHASH_DOALL_ARG(const void *, const void *);
#define IMPLEMENT_LHASH_DOALL_ARG_FN(f_name,o_type,a_type) \
        void f_name##_LHASH_DOALL_ARG(const void *arg1, const void *arg2) { \
                o_type a = (o_type)arg1; \
                a_type b = (a_type)arg2; \
                f_name(a,b); }
#define LHASH_DOALL_ARG_FN(f_name) f_name##_LHASH_DOALL_ARG
```

An example of a hash table storing (pointers to) structures of type 'STUFF' could be defined as follows;

```
/* Calculates the hash value of 'tohash' (implemented elsewhere) */
unsigned long STUFF_hash(const STUFF *tohash);
/* Orders 'arg1' and 'arg2' (implemented elsewhere) */
int STUFF_cmp(const STUFF *arg1, const STUFF *arg2);
/* Create the type-safe wrapper functions for use in the LHASH internals */
static IMPLEMENT_LHASH_HASH_FN(STUFF_hash, const STUFF *)
static IMPLEMENT_LHASH_COMP_FN(STUFF_cmp, const STUFF *);
/* ... */
int main(int argc, char *argv[]) {
        /* Create the new hash table using the hash/compare wrappers */
        LHASH *hashtable = lh_new(LHASH_HASH_FN(STUFF_hash),
                                  LHASH_COMP_FN(STUFF_cmp));
/* ... */
}
```

lh_free() frees the *LHASH* structure *table*. Allocated hash table entries will not be freed; consider using lh_doall() to deallocate any remaining entries in the hash table (see below).

lh_insert() inserts the structure pointed to by *data* into *table*. If there already is an entry with the same key, the old value is replaced. Note that lh_insert() stores pointers, the data are not copied.

lh_delete() deletes an entry from *table*.

lh_retrieve() looks up an entry in *table*. Normally, *data* is a structure with the key field(s) set; the function will return a pointer to a fully populated structure.

lh_doall() will, for every entry in the hash table, call *func* with the data item as its parameter. For lh_doall() and lh_doall_arg(), function pointer casting should be avoided in the callbacks (see *NOTE*) - instead, either declare the callbacks to match the prototype required in lh_new() or use the declare/implement macros to create type-safe wrappers that cast variables prior to calling your type-specific callbacks. An example of this is illustrated here where the callback is used to cleanup resources for items in the hash table prior to the hashtable itself being deallocated:

```
/* Cleans up resources belonging to 'a' (this is implemented elsewhere) */
void STUFF_cleanup(STUFF *a);
/* Implement a prototype-compatible wrapper for "STUFF_cleanup" */
IMPLEMENT_LHASH_DOALL_FN(STUFF_cleanup, STUFF *)
        /* ... then later in the code ... */
/* So to run "STUFF_cleanup" against all items in a hash table ... */
```

```
lh_doall(hashtable, LHASH_DOALL_FN(STUFF_cleanup));
/* Then the hash table itself can be deallocated */
lh_free(hashtable);
```

When doing this, be careful if you delete entries from the hash table in your callbacks: the table may decrease in size, moving the item that you are currently on down lower in the hash table - this could cause some entries to be skipped during the iteration. The second best solution to this problem is to set hash->down_load=0 before you start (which will stop the hash table ever decreasing in size). The best solution is probably to avoid deleting items from the hash table inside a "doall" callback!

lh_doall_arg() is the same as lh_doall() except that *func* will be called with *arg* as the second argument and *func* should be of type *LHASH_DOALL_ARG_FN_TYPE* (a callback prototype that is passed both the table entry and an extra argument). As with lh_doall(), you can instead choose to declare your callback with a prototype matching the types you are dealing with and use the declare/implement macros to create compatible wrappers that cast variables before calling your type-specific callbacks. An example of this is demonstrated here (printing all hash table entries to a BIO that is provided by the caller):

```
/* Prints item 'a' to 'output_bio' (this is implemented elsewhere) */
void STUFF_print(const STUFF *a, BIO *output_bio);
/* Implement a prototype-compatible wrapper for "STUFF_print" */
static IMPLEMENT_LHASH_DOALL_ARG_FN(STUFF_print, const STUFF *, BIO *)
        /* ... then later in the code ... */
/* Print out the entire hashtable to a particular BIO */
lh_doall_arg(hashtable, LHASH_DOALL_ARG_FN(STUFF_print), logging_bio);
```

```
lh_error() can be used to determine if an error occurred in the last
operation. lh_error() is a macro.
```

RETURN VALUES

lh_new() returns *NULL* on error, otherwise a pointer to the new *LHASH* structure.

When a hash table entry is replaced, lh_insert() returns the value being replaced. *NULL* is returned on normal operation and on error.

lh_delete() returns the entry being deleted. *NULL* is returned if there is no such value in the hash table.

lh_retrieve() returns the hash table entry if it has been found, *NULL* otherwise.

lh_error() returns 1 if an error occurred in the last operation, 0 otherwise.

lh_free(), lh_doall() and lh_doall_arg() return no values.

NOTE

The various LHASH macros and callback types exist to make it possible to write type-safe code without resorting to function-prototype casting - an evil that makes application code much harder to audit/verify and also opens the window of opportunity for stack corruption and other hard-to-find bugs. It also, apparently, violates ANSI-C.

The LHASH code regards table entries as constant data. As such, it internally represents lh_insert()'d items with a "const void *" pointer type. This is why callbacks such as those used by lh_doall() and lh_doall_arg() declare their prototypes with "const", even for the parameters that pass back the table items' data pointers - for consistency, user-provided data is "const" at all times as far as the LHASH code is concerned. However, as callers are themselves providing these pointers, they can choose whether they too should be treating all such parameters as constant.

As an example, a hash table may be maintained by code that, for reasons of encapsulation, has only "const" access to the data being indexed in the hash table (ie. it is returned as "const" from elsewhere in their code) - in this case the LHASH prototypes are appropriate as-is. Conversely, if the caller is responsible for the life-time of the data in question, then they may well wish to make modifications to table item passed back in the lh_doall() or lh_doall_arg() callbacks (see the "STUFF_cleanup" example above). If so, the caller can either cast the "const" away (if they're providing the raw callbacks themselves) or use the macros to declare/implement the wrapper functions without "const" types.

Callers that only have "const" access to data they're indexing in a table, yet declare callbacks without constant types (or cast the "const" away themselves), are therefore creating their own risks/bugs without being encouraged to do so by the API. On a related note, those auditing code should pay special attention to any instances of DECLARE/IMPLEMENT_LHASH_DOALL_[ARG_]_FN macros that provide types without any "const" qualifiers.

Restrictions

lh_insert() returns *NULL* both for success and error.

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 OPENSSL_realloc(). 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 just playing with 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.

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 lh_new().

SEE ALSO

lh_stats (3)

HISTORY

The *lhash* library is available in all versions of SSLeay and OpenSSL. lh_error() was added in SSLeay 0.9.1b.

This manpage is derived from the SSLeay documentation.

In OpenSSL 0.9.7, all lhash functions that were passed function pointers were changed for better type safety, and the function types LHASH_COMP_FN_TYPE, LHASH_HASH_FN_TYPE, LHASH_DOALL_FN_TYPE and LHASH_DOALL_ARG_FN_TYPE became available.

MD2

NAME

MD2, MD4, MD5, MD2_Init, MD2_Update, MD2_Final, MD4_Init, MD4_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.

MD2(), MD4(), and MD5() 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:

MD2_Init() initializes a *MD2_CTX* structure.

MD2_Update() can be called repeatedly with chunks of the message to be hashed (*len* bytes at *data*).

MD2_Final() places the message digest in *md*, which must have space for MD2_DIGEST_LENGTH == 16 bytes of output, and erases the *MD2_CTX*.

MD4_Init(), MD4_Update(), MD4_Final(), MD5_Init(), MD5_Update(), and MD5_Final() are analogous using an *MD4_CTX* and *MD5_CTX* structure.

Applications should use the higher level functions *EVP_DigestInit* (3) etc. instead of calling the hash functions directly.

NOTE

MD2, MD4, and MD5 are recommended only for compatibility with existing applications. In new applications, SHA-1 or RIPEMD-160 should be preferred.

RETURN VALUES

MD2(), MD4(), and MD5() return pointers to the hash value.

MD2_Init(), MD2_Update(), MD2_Final(), MD4_Init(), MD4_Update(), MD4_Final(), MD5_Init(), MD5_Update(), and MD5_Final() do not return values.

CONFORMING TO

RFC 1319, RFC 1320, RFC 1321

SEE ALSO

sha (3), *ripemd* (3), *EVP_DigestInit* (3)

HISTORY

MD2(), MD2_Init(), MD2_Update() MD2_Final(), MD5(), MD5_Init(), MD5_Update() and MD5_Final() are available in all versions of SSLeay and OpenSSL.

MD4(), MD4_Init(), and MD4_Update() are available in OpenSSL 0.9.6 and above.

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.

MDC2() 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 may be used if the message is not completely stored in memory:

MDC2_Init() initializes a *MDC2_CTX* structure.

MDC2_Update() can be called repeatedly with chunks of the message to be hashed (*len* bytes at *data*).

MDC2_Final() 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 *EVP_DigestInit* (3) etc. instead of calling the hash functions directly.

RETURN VALUES

MDC2() returns a pointer to the hash value.

MDC2_Init(), MDC2_Update() and MDC2_Final() do not return values.

CONFORMING TO

ISO/IEC 10118-2, with DES

SEE ALSO

sha (3), *EVP_DigestInit* (3)

HISTORY

MDC2(), MDC2_Init(), MDC2_Update() and MDC2_Final() are available since SSLeay 0.8.

OBJ_nid2obj

NAME

OBJ_nid2obj, OBJ_nid2ln, OBJ_nid2sn, OBJ_obj2nid, OBJ_txt2nid, OBJ_ln2nid, OBJ_sn2nid,

OBJ_cmp, OBJ_dup, OBJ_txt2obj, OBJ_obj2txt, OBJ_create, OBJ_cleanup – ASN1 object utility functions

Synopsis

```
ASN1_OBJECT * OBJ_nid2obj(int n);
const char * OBJ_nid2ln(int n);
const char * OBJ_nid2sn(int n);
int OBJ_obj2nid(const ASN1_OBJECT *o);
int OBJ_ln2nid(const char *ln);
int OBJ_sn2nid(const char *sn);
int OBJ_txt2nid(const char *s);
ASN1_OBJECT * OBJ_txt2obj(const char *s, int no_name);
int OBJ_obj2txt(char *buf, int buf_len, const ASN1_OBJECT *a, int no_name);
int OBJ_cmp(const ASN1_OBJECT *a,const ASN1_OBJECT *b);
ASN1_OBJECT * OBJ_dup(const ASN1_OBJECT *o);
int OBJ_create(const char *oid,const char *sn,const char *ln);
void OBJ_cleanup(void);
```

DESCRIPTION

The ASN1 object utility functions process ASN1_OBJECT structures which are a representation of the ASN1 OBJECT IDENTIFIER (OID) type.

OBJ_nid2obj(), OBJ_nid2ln() and OBJ_nid2sn() convert the NID *n* to an ASN1_OBJECT structure, its long name and its short name respectively, or *NULL* is an error occurred.

OBJ_obj2nid(), OBJ_ln2nid(), OBJ_sn2nid() return the corresponding NID for the object *o*, the long name <ln> or the short name <sn> respectively or NID_undef if an error occurred.

OBJ_txt2nid() returns NID corresponding to text string <s>. *s* can be a long name, a short name or the numerical respresentation of an object.

OBJ_txt2obj() converts the text string *s* into an ASN1_OBJECT structure. If *no_name* is 0 then long names and short names will be interpreted as well as numerical forms. If *no_name* is 1 only the numerical form is acceptable.

OBJ_obj2txt() converts the *ASN1_OBJECT a* into a textual representation. The representation is written as a null terminated string to *buf* at most *buf_len* bytes are written, truncating the result if necessary. The total amount of space required is returned. If *no_name* is 0 then if the object has a long or short name then that will be used, otherwise the numerical form will be used. If *no_name* is 1 then the numerical form will always be used.

OBJ_cmp() compares *a* to *b*. If the two are identical 0 is returned.

OBJ_dup() returns a copy of *o*.

OBJ_create() adds a new object to the internal table. *oid* is the numerical form of the object, *sn* the short name and *ln* the long name. A new NID is returned for the created object.

OBJ_cleanup() cleans up OpenSSLs internal object table: this should be called before an application exits if any new objects were added using OBJ_create().

NOTES

Objects in OpenSSL can have a short name, a long name and a numerical identifier (NID) associated with them. A standard set of objects is represented in an internal table. The appropriate values are defined in the header file *objects.h*.

For example the OID for commonName has the following definitions:

```
#define SN_commonName                   "CN"
#define LN_commonName                   "commonName"
#define NID_commonName                  13
```

New objects can be added by calling OBJ_create().

Table objects have certain advantages over other objects: for example their NIDs can be used in a C language switch statement. They are also static constant structures which are shared: that is there is only a single constant structure for each table object.

Objects which are not in the table have the NID value NID_undef.

Objects do not need to be in the internal tables to be processed, the functions OBJ_txt2obj() and OBJ_obj2txt() can process the numerical form of an OID.

EXAMPLES

Create an object for *commonName*:

```
ASN1_OBJECT *o;
o = OBJ_nid2obj(NID_commonName);
```

Check if an object is *commonName*

```
if (OBJ_obj2nid(obj) == NID_commonName)
/* Do something */
```

Create a new NID and initialize an object from it:

```
int new_nid;
ASN1_OBJECT *obj;
new_nid = OBJ_create("1.2.3.4", "NewOID", "New Object Identifier");

obj = OBJ_nid2obj(new_nid);

Create a new object directly:

obj = OBJ_txt2obj("1.2.3.4", 1);
```

Restrictions

OBJ_obj2txt() is awkward and messy to use: it doesn't follow the convention of other OpenSSL functions where the buffer can be set to *NULL* to determine the amount of data that should be written. Instead *buf* must point to a valid buffer and *buf_len* should be set to a positive value. A buffer length of 80 should be more than enough to handle any OID encountered in practice.

RETURN VALUES

OBJ_nid2obj() returns an *ASN1_OBJECT* structure or *NULL* is an error occurred.

OBJ_nid2ln() and OBJ_nid2sn() returns a valid string or *NULL* on error.

OBJ_obj2nid(), OBJ_ln2nid(), OBJ_sn2nid() and OBJ_txt2nid() return a NID or *NID_undef* on error.

SEE ALSO

ERR_get_error (3)

HISTORY

None.

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().

OpenSSL_add_all_digests() adds all digest algorithms to the table.

OpenSSL_add_all_algorithms() adds all algorithms to the table (digests and ciphers).

OpenSSL_add_all_ciphers() adds all encryption algorithms to the table including password based encryption algorithms.

EVP_cleanup() removes all ciphers and digests from the table.

RETURN VALUES

None of the functions return a value.

NOTES

A typical application will will call OpenSSL_add_all_algorithms() initially and EVP_cleanup() before exiting.

An application does not need to add algorithms to use them explicitly, for example by EVP_sha1(). It just 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 misbehave 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 OpenSSL_add_all_algorithms() links in all algorithms: as a result a statically linked executable can be quite large. If this is important it is possible to just add 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 this is not too much of a problem in practice.

SEE ALSO

evp (3), *EVP_DigestInit* (3), *EVP_EncryptInit* (3)

OPENSSL_config

NAME

OPENSSL_config, OPENSSL_no_config – simple OpenSSL configuration functions

Synopsis

```
#include <openssl/conf.h>
void OPENSSL_config(const char *config_name);
void OPENSSL_no_config(void);
```

DESCRIPTION

OPENSSL_config() configures OpenSSL using the standard *openssl.cnf* configuration file name using *config_name*. If *config_name* is NULL then the default name *openssl_conf* will be used. Any errors are ignored. Further calls to OPENSSL_config() will have no effect. The configuration file format is documented in the *conf* (5) manual page.

OPENSSL_no_config() disables configuration. If called before OPENSSL_config() no configuration takes place.

NOTES

It is *strongly* recommended that *all* new applications call OPENSSL_config() or the more sophisticated functions such as CONF_modules_load() during initialization (that is before starting any threads). By doing this an application does not need to keep track of all configuration options and some new functionality can be supported automatically.

It is also possible to automatically call OPENSSL_config() when an application calls OPENSSL_add_all_algorithms() by compiling an application with the preprocessor symbol *OPENSSL_LOAD_CONF* #define'd. In this way configuration can be added without source changes.

The environment variable *OPENSSL_CONFIG* can be set to specify the location of the configuration file. Currently ASN1 OBJECTs and ENGINE configuration can be performed future versions of OpenSSL will add new configuration options.

There are several reasons why calling the OpenSSL configuration routines is advisable. For example new ENGINE functionality was added to OpenSSL 0.9.7. In OpenSSL 0.9.7 control functions can be supported by ENGINEs, this can be used (among other things) to load dynamic ENGINEs from shared libraries (DSOs). However very few applications currently support the control interface and so very few can load and use dynamic ENGINEs. Equally in future more sophisticated ENGINEs will require certain control operations to customize them. If an application calls OPENSSL_config() it doesn't need to know or care about ENGINE control operations because they can be performed by editing a configuration file.

Applications should free up configuration at application closedown by calling CONF_modules_free().

RESTRICTIONS

The OPENSSL_config() function is designed to be a very simple "call it and forget it" function. As a result its behaviour is somewhat limited. It ignores all errors silently and it can only load from the standard configuration file location for example.

It is however *much* better than nothing. Applications which need finer control over their configuration functionality should use the configuration functions such as CONF_load_modules() directly.

RETURN VALUES

Neither OPENSSL_config() nor OPENSSL_no_config() return a value.

SEE ALSO

conf (5), *CONF_load_modules_file* (3), *CONF_modules_free* (3), *CONF_modules_free* (3)

HISTORY

OPENSSL_config() and OPENSSL_no_config() first appeared in OpenSSL 0.9.7

OPENSSL_load_builtin_modules

NAME

OPENSSL_load_builtin_modules – add standard configuration modules

Synopsis

```
#include <openssl/conf.h>
void OPENSSL_load_builtin_modules(void);
void ASN1_add_oid_module(void);
ENGINE_add_conf_module();
```

DESCRIPTION

The function OPENSSL_load_builtin_modules() adds all the standard OpenSSL configuration modules to the internal list. They can then be used by the OpenSSL configuration code.

ASN1_add_oid_module() adds just the ASN1 OBJECT module.

ENGINE_add_conf_module() adds just the ENGINE configuration module.

NOTES

If the simple configuration function OPENSSL_config() is called then OPENSSL_load_builtin_modules() is called automatically.

Applications which use the configuration functions directly will need to call OPENSSL_load_builtin_modules() themselves *before* any other configuration code.

Applications should call OPENSSL_load_builtin_modules() to load all configuration modules instead of adding modules selectively: otherwise functionality may be missing from the application if an when new modules are added.

RETURN VALUE

None of the functions return a value.

SEE ALSO

conf (3), *OPENSSL_config* (3)

HISTORY

These functions first appeared in OpenSSL 0.9.7.

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);
const 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
```

for example

```
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.

SSLeay() 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.

SSLeay_version() 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 compiler flags set for the compilation process in the form "compiler: ..." if available or "compiler: information not available" otherwise.

- SSLEAY_BUILT_ON

  The date of the build process in the form "built on: ..." if available or "built on: date not available" otherwise.

- SSLEAY_PLATFORM

The "Configure" target of the library build in the form "platform: ..." if available or "platform: information not available" otherwise.

- SSLEAY_DIR

  The "OPENSSLDIR" setting of the library build in the form "OPENSSLDIR: "..."" if available or "OPENSSLDIR: N/A" otherwise.

For an unknown *t*, the text "not available" is returned.

RETURN VALUE

The version number.

SEE ALSO

crypto (3)

HISTORY

SSLeay() and SSLEAY_VERSION_NUMBER are available in all versions of SSLeay and OpenSSL. OPENSSL_VERSION_NUMBER is available in all versions of OpenSSL. *SSLEAY_DIR* was added in OpenSSL 0.9.7.

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, unsigned char
*kstr, int klen, 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_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 *PEM_read_bio_X509(BIO *bp, X509 **x, pem_password_cb *cb, void *u);
X509 *PEM_read_X509(FILE *fp, X509 **x, pem_password_cb *cb, void *u);
int PEM_write_bio_X509(BIO *bp, X509 *x);
int PEM_write_X509(FILE *fp, X509 *x);
X509 *PEM_read_bio_X509_AUX(BIO *bp, X509 **x, pem_password_cb *cb, void *u);
X509 *PEM_read_X509_AUX(FILE *fp, X509 **x, pem_password_cb *cb, void *u);
int PEM_write_bio_X509_AUX(BIO *bp, X509 *x);
int PEM_write_X509_AUX(FILE *fp, X509 *x);
X509_REQ *PEM_read_bio_X509_REQ(BIO *bp, X509_REQ **x, pem_password_cb *cb, void *u);
X509_REQ *PEM_read_X509_REQ(FILE *fp, X509_REQ **x, pem_password_cb *cb, void *u);
int PEM_write_bio_X509_REQ(BIO *bp, X509_REQ *x);
int PEM_write_X509_REQ(FILE *fp, X509_REQ *x);
int PEM_write_bio_X509_REQ_NEW(BIO *bp, X509_REQ *x);
int PEM_write_X509_REQ_NEW(FILE *fp, X509_REQ *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.

For more details about the meaning of arguments see the *PEM FUNCTION ARGUMENTS* section.

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 additionally transparently handle PKCS#8 format encrypted and unencrypted keys too.

PEM_write_bio_PKCS8PrivateKey() and PEM_write_PKCS8PrivateKey() 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.

PEM_write_bio_PKCS8PrivateKey_nid() and PEM_write_PKCS8PrivateKey_nid() 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* BIO 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* (but see Restrictions 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 pass phrase 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* parameters 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.

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)
{
/* Error */
}
```

Alternative method:

```
X509 *x = NULL;
if (!PEM_read_bio_X509(bp, &x, 0, NULL))
{
/* Error */
}
```

Write a certificate to a BIO:

```
if (!PEM_write_bio_X509(bp, x))
{
/* Error */
}
```

Write an unencrypted private key to a FILE pointer:

```
if (!PEM_write_PrivateKey(fp, key, NULL, NULL, 0, 0, NULL))
{
/* Error */
}
```

Write a private key (using traditional format) to a BIO using triple DES encryption, the pass phrase is prompted for:

```
 if (!PEM_write_bio_PrivateKey(bp, key, EVP_des_ede3_cbc(), NULL, 0, 0, NULL))
{
/* Error */
}
```

Write a private key (using PKCS#8 format) to a BIO using triple DES encryption, using the pass phrase "hello":

```
 if (!PEM_write_bio_PKCS8PrivateKey(bp, key, EVP_des_ede3_cbc(), NULL, 0, 0, "hello"))
{
/* Error */
}
```

Read a private key from a BIO using the pass phrase "hello":

```
 key = PEM_read_bio_PrivateKey(bp, NULL, 0, "hello");
 if (key == NULL)
{
/* Error */
}
```

Read a private key from a BIO using a pass phrase callback:

```
 key = PEM_read_bio_PrivateKey(bp, NULL, pass_cb, "My Private Key");
 if (key == NULL)
{
/* Error */
}
```

Skeleton pass phrase callback:

```
 int pass_cb(char *buf, int size, int rwflag, void *u);
{
int len;
char *tmp;
/* We'd probably do something else if 'rwflag' is 1 */
printf("Enter pass phrase for \"%s\"\n", u);

/* get pass phrase, length 'len' into 'tmp' */
tmp = "hello";
len = strlen(tmp);

if (len <= 0) return 0;
/* if too long, truncate */
if (len > size) len = size;
memcpy(buf, tmp, len);
return len;
}
```

NOTES

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 (they use an iteration count of 2048 whereas the traditional routines use a count of 1) unless compatibility with older versions of OpenSSL is important.

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 like this:

```
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 uninitialised pointer.

PEM ENCRYPTION FORMAT

This old *PrivateKey* routines use a non standard technique for encryption.

The private key (or other data) takes the following form:

```
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: DES-EDE3-CBC,3F17F5316E2BAC89

...base64 encoded data...
-----END RSA PRIVATE KEY-----
```

The line beginning DEK-Info contains two comma separated pieces of information: the encryption algorithm name as used by EVP_get_cipherbyname() and an 8 byte *salt* encoded as a set of hexadecimal digits.

After this is the base64 encoded encrypted data.

The encryption key is determined using EVP_bytestokey(), using *salt* and an iteration count of 1. The IV used is the value of *salt* and *not* the IV returned by EVP_bytestokey().

Restrictions

The PEM read routines in some versions of OpenSSL will not correctly reuse an existing structure. Therefore the following:

```
PEM_read_bio(bp, &x, 0, NULL);
```

where *x* already contains a valid certificate, may not work, whereas:

```
X509_free(x);
x = PEM_read_bio(bp, NULL, 0, NULL);
```

is guaranteed to work.

RETURN CODES

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.

PKCS12_create

NAME

PKCS12_create – create a PKCS#12 structure

Synopsis

```
#include <openssl/pkcs12.h>
PKCS12 *PKCS12_create(char *pass, char *name, EVP_PKEY *pkey, X509 *cert, STACK_OF(X509)
*ca, int nid_key, int nid_cert, int iter, int mac_iter, int keytype);
```

DESCRIPTION

PKCS12_create() creates a PKCS#12 structure.

pass is the passphrase to use. *name* is the *friendlyName* to use for the supplied certifictate and key. *pkey* is the private key to include in the structure and *cert* its corresponding certificates. *ca*, if not *NULL* is an optional set of certificates to also include in the structure.

nid_key and *nid_cert* are the encryption algorithms that should be used for the key and certificate respectively. *iter* is the encryption algorithm iteration count to use and *mac_iter* is the MAC iteration count to use. *keytype* is the type of key.

NOTES

The parameters *nid_key*, *nid_cert*, *iter*, *mac_iter* and *keytype* can all be set to zero and sensible defaults will be used.

These defaults are: 40 bit RC2 encryption for certificates, triple DES encryption for private keys, a key iteration count of PKCS12_DEFAULT_ITER (currently 2048) and a MAC iteration count of 1.

The default MAC iteration count is 1 in order to retain compatibility with old software which did not interpret MAC iteration counts. If such compatibility is not required then *mac_iter* should be set to PKCS12_DEFAULT_ITER.

keytype adds a flag to the store private key. This is a non standard extension that is only currently interpreted by MSIE. If set to zero the flag is omitted, if set to *KEY_SIG* the key can be used for signing only, if set to *KEY_EX* it can be used for signing and encryption. This option was useful for old export grade software which could use signing only keys of arbitrary size but had restrictions on the permissible sizes of keys which could be used for encryption.

SEE ALSO

d2i_PKCS12(3)

HISTORY

PKCS12_create was added in OpenSSL 0.9.3

PKCS12_parse

NAME

PKCS12_parse – parse a PKCS#12 structure

Synopsis

```
#include <openssl/pkcs12.h>
int PKCS12_parse(PKCS12 *p12, const char *pass, EVP_PKEY **pkey, X509 **cert,
STACK_OF(X509) **ca);
```

DESCRIPTION

PKCS12_parse() parses a PKCS12 structure.

p12 is the *PKCS12* structure to parse. *pass* is the passphrase to use. If successful the private key will be written to **pkey*, the corresponding certificate to **cert* and any additional certificates to **ca*.

NOTES

The parameters *pkey* and *cert* cannot be *NULL*. *ca* can be <NULL> in which case additional certificates will be discarded. **ca* can also be a valid STACK in which case additional certificates are appended to **ca*. If **ca* is *NULL* a new STACK will be allocated.

The *friendlyName* and *localKeyID* attributes (if present) on each certificate will be stored in the *alias* and *keyid* attributes of the *X509* structure.

Restrictions

Only a single private key and corresponding certificate is returned by this function. More complex PKCS#12 files with multiple private keys will only return the first match.

Only *friendlyName* and *localKeyID* attributes are currently stored in certificates. Other attributes are discarded.

Attributes currently cannot be store in the private key *EVP_PKEY* structure.

SEE ALSO

d2i_PKCS12(3)

HISTORY

PKCS12_parse was added in OpenSSL 0.9.3

PKCS7_decrypt

NAME

PKCS7_decrypt – decrypt content from a PKCS#7 envelopedData structure

Synopsis

```
int PKCS7_decrypt(PKCS7 *p7, EVP_PKEY *pkey, X509 *cert, BIO *data, int flags);
```

DESCRIPTION

PKCS7_decrypt() extracts and decrypts the content from a PKCS#7 envelopedData structure. *pkey* is the private key of the recipient, *cert* is the recipients certificate, *data* is a BIO to write the content to and *flags* is an optional set of flags.

NOTES

OpenSSL_add_all_algorithms() (or equivalent) should be called before using this function or errors about unknown algorithms will occur.

Although the recipients certificate is not needed to decrypt the data it is needed to locate the appropriate (of possible several) recipients in the PKCS#7 structure.

The following flags can be passed in the *flags* parameter.

If the *PKCS7_TEXT* flag is set MIME headers for type *text/plain* are deleted from the content. If the content is not of type *text/plain* then an error is returned.

RETURN VALUES

PKCS7_decrypt() returns either 1 for success or 0 for failure. The error can be obtained from *ERR_get_error* (3)

Restrictions

PKCS7_decrypt() must be passed the correct recipient key and certificate. It would be better if it could look up the correct key and certificate from a database.

The lack of single pass processing and need to hold all data in memory as mentioned in PKCS7_sign() also applies to PKCS7_verify().

SEE ALSO

ERR_get_error (3), PKCS7_*encrypt* (3)

HISTORY

PKCS7_decrypt() was added to OpenSSL 0.9.5

PKCS7_encrypt

NAME

PKCS7_encrypt – create a PKCS#7 envelopedData structure

Synopsis

```
PKCS7 *PKCS7_encrypt(STACK_OF(X509) *certs, BIO *in, const EVP_CIPHER *cipher, int flags);
```

DESCRIPTION

PKCS7_encrypt() creates and returns a PKCS#7 envelopedData structure. *certs* is a list of recipient certificates. *in* is the content to be encrypted. *cipher* is the symmetric cipher to use. *flags* is an optional set of flags.

NOTES

Only RSA keys are supported in PKCS#7 and envelopedData so the recipient certificates supplied to this function must all contain RSA public keys, though they do not have to be signed using the RSA algorithm.

EVP_des_ede3_cbc() (triple DES) is the algorithm of choice for S/MIME use because most clients will support it.

Some old "export grade" clients may only support weak encryption using 40 or 64 bit RC2. These can be used by passing EVP_rc2_40_cbc() and EVP_rc2_64_cbc() respectively.

The algorithm passed in the *cipher* parameter must support ASN1 encoding of its parameters.

Many browsers implement a "sign and encrypt" option which is simply an S/MIME envelopedData containing an S/MIME signed message. This can be readily produced by storing the S/MIME signed message in a memory BIO and passing it to PKCS7_encrypt().

The following flags can be passed in the *flags* parameter.

If the *PKCS7_TEXT* flag is set MIME headers for type *text/plain* are prepended to the data.

Normally the supplied content is translated into MIME canonical format (as required by the S/MIME specifications) if *PKCS7_BINARY* is set no translation occurs. This option should be used if the supplied data is in binary format otherwise the translation will corrupt it. If *PKCS7_BINARY* is set then *PKCS7_TEXT* is ignored.

RETURN VALUES

PKCS7_encrypt() returns either a valid PKCS7 structure or NULL if an error occurred. The error can be obtained from *ERR_get_error* (3).

Restrictions

The lack of single pass processing and need to hold all data in memory as mentioned in PKCS7_sign() also applies to PKCS7_verify().

SEE ALSO

ERR_get_error (3), PKCS7_*decrypt* (3)

HISTORY

PKCS7_decrypt() was added to OpenSSL 0.9.5

PKCS7_sign

NAME

PKCS7_sign – create a PKCS#7 signedData structure

Synopsis

```
PKCS7 *PKCS7_sign(X509 *signcert, EVP_PKEY *pkey, STACK_OF(X509) *certs, BIO *data, int
flags);
```

DESCRIPTION

PKCS7_sign() creates and returns a PKCS#7 signedData structure. *signcert* is the certificate to sign with, *pkey* is the corresponsding private key. *certs* is an optional additional set of certificates to include in the PKCS#7 structure (for example any intermediate CAs in the chain).

The data to be signed is read from BIO *data*.

flags is an optional set of flags.

NOTES

Any of the following flags (ored together) can be passed in the *flags* parameter.

Many S/MIME clients expect the signed content to include valid MIME headers. If the *PKCS7_TEXT* flag is set MIME headers for type *text/plain* are prepended to the data.

If *PKCS7_NOCERTS* is set the signer's certificate will not be included in the PKCS7 structure, the signer's certificate must still be supplied in the *signcert* parameter though. This can reduce the size of the signature if the signers certificate can be obtained by other means: for example a previously signed message.

The data being signed is included in the PKCS7 structure, unless *PKCS7_DETACHED* is set in which case it is omitted. This is used for PKCS7 detached signatures which are used in S/MIME plaintext signed messages for example.

Normally the supplied content is translated into MIME canonical format (as required by the S/MIME specifications) if *PKCS7_BINARY* is set no translation occurs. This option should be used if the supplied data is in binary format otherwise the translation will corrupt it.

The signedData structure includes several PKCS#7 autenticatedAttributes including the signing time, the PKCS#7 content type and the supported list of ciphers in an SMIMECapabilities attribute. If *PKCS7_NOATTR* is set then no authenticatedAttributes will be used. If *PKCS7_NOSMIMECAP* is set then just the SMIMECapabilities are omitted.

If present the SMIMECapabilities attribute indicates support for the following algorithms: triple DES, 128 bit RC2, 64 bit RC2, DES and 40 bit RC2. If any of these algorithms is disabled then it will not be included.

Restrictions

PKCS7_sign() is somewhat limited. It does not support multiple signers, some advanced attributes such as counter signatures are not supported.

The SHA1 digest algorithm is currently always used.

When the signed data is not detached it will be stored in memory within the *PKCS7* structure. This effectively limits the size of messages which can be signed due to memory restraints. There should be a way to sign data without having to hold it all in memory, this would however require fairly major revisions of the OpenSSL ASN1 code.

Clear text signing does not store the content in memory but the way PKCS7_sign() operates means that two passes of the data must typically be made: one to compute the signatures and a second to output the data along with the signature. There should be a way to process the data with only a single pass.

RETURN VALUES

PKCS7_sign() returns either a valid PKCS7 structure or NULL if an error occurred. The error can be obtained from *ERR_get_error* (3).

SEE ALSO

ERR_get_error (3), PKCS7_*verify* (3)

HISTORY

PKCS7_sign() was added to OpenSSL 0.9.5

PKCS7_verify

NAME

PKCS7_verify – verify a PKCS#7 signedData structure

Synopsis

```
int PKCS7_verify(PKCS7 *p7, STACK_OF(X509) *certs, X509_STORE *store, BIO *indata, BIO
*out, int flags);
int PKCS7_get0_signers(PKCS7 *p7, STACK_OF(X509) *certs, int flags);
```

DESCRIPTION

PKCS7_verify() verifies a PKCS#7 signedData structure. *p7* is the PKCS7 structure to verify. *certs* is a set of certificates in which to search for the signer's certificate. *store* is a trusted certficate store (used for chain verification). *indata* is the signed data if the content is not present in *p7* (that is it is detached). The content is written to *out* if it is not NULL.

flags is an optional set of flags, which can be used to modify the verify operation.

PKCS7_get0_signers() retrieves the signer's certificates from *p7*, it does *not* check their validity or whether any signatures are valid. The *certs* and *flags* parameters have the same meanings as in PKCS7_verify().

VERIFY PROCESS

Normally the verify process proceeds as follows.

Initially some sanity checks are performed on *p7*. The type of *p7* must be signedData. There must be at least one signature on the data and if the content is detached *indata* cannot be *NULL*.

An attempt is made to locate all the signer's certificates, first looking in the *certs* parameter (if it is not *NULL*) and then looking in any certificates contained in the *p7* structure itself. If any signer's certificates cannot be located the operation fails.

Each signer's certificate is chain verified using the *smimesign* purpose and the supplied trusted certificate store. Any internal certificates in the message are used as untrusted CAs. If any chain verify fails an error code is returned.

Finally the signed content is read (and written to *out* is it is not NULL) and the signature's checked.

If all signature's verify correctly then the function is successful.

Any of the following flags (ored together) can be passed in the *flags* parameter to change the default verify behaviour. Only the flag *PKCS7_NOINTERN* is meaningful to PKCS7_get0_signers().

If *PKCS7_NOINTERN* is set the certificates in the message itself are not searched when locating the signer's certificate. This means that all the signers certificates must be in the *certs* parameter.

If the *PKCS7_TEXT* flag is set MIME headers for type *text/plain* are deleted from the content. If the content is not of type *text/plain* then an error is returned.

If *PKCS7_NOVERIFY* is set the signer's certificates are not chain verified.

If *PKCS7_NOCHAIN* is set then the certificates contained in the message are not used as untrusted CAs. This means that the whole verify chain (apart from the signer's certificate) must be contained in the trusted store.

If *PKCS7_NOSIGS* is set then the signatures on the data are not checked.

NOTES

One application of *PKCS7_NOINTERN* is to only accept messages signed by a small number of certificates. The acceptable certificates would be passed in the *certs* parameter. In this case if the signer is not one of the certificates supplied in *certs* then the verify will fail because the signer cannot be found.

Care should be taken when modifying the default verify behaviour, for example setting *PKCS7_NOVERIFY|PKCS7_NOSIGS* will totally disable all verification and any signed message will be considered valid. This combination is however useful if one merely wishes to write the content to *out* and its validity is not considered important.

Chain verification should arguably be performed using the signing time rather than the current time. However since the signing time is supplied by the signer it cannot be trusted without additional evidence (such as a trusted timestamp).

RETURN VALUES

PKCS7_verify() returns 1 for a successful verification and zero or a negative value if an error occurs.

PKCS7_get0_signers() returns all signers or *NULL* if an error occurred.

The error can be obtained from *ERR_get_error* (3)

Restrictions

The trusted certificate store is not searched for the signers certificate, this is primarily due to the inadequacies of the current *X509_STORE* functionality.

The lack of single pass processing and need to hold all data in memory as mentioned in PKCS7_sign() also applies to PKCS7_verify().

SEE ALSO

ERR_get_error (3), PKCS7_*sign* (3)

HISTORY

PKCS7_verify() was added to OpenSSL 0.9.5

rand

NAME

rand – pseudo-random number generator

Synopsis

```
#include <openssl/rand.h>
int RAND_set_rand_engine(ENGINE *engine);
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);
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(const RAND_METHOD *meth);
const RAND_METHOD *RAND_get_rand_method(void);
RAND_METHOD *RAND_SSLeay(void);
void RAND_cleanup(void);
/* For Win32 only */ void RAND_screen(void);
int RAND_event(UINT, WPARAM, LPARAM);
```

DESCRIPTION

Since the introduction of the ENGINE API, the recommended way of controlling default implementations is by using the ENGINE API functions. The default *RAND_METHOD*, as set by RAND_set_rand_method() and returned by RAND_get_rand_method(), is only used if no ENGINE has been set as the default "rand" implementation. Hence, these two functions are no longer the recommened way to control defaults.

If an alternative *RAND_METHOD* implementation is being used (either set directly or as provided by an ENGINE module), then it is entirely responsible for the generation and management of a cryptographically secure PRNG stream. The mechanisms described below relate solely to the software PRNG implementation built in to OpenSSL and used by default.

These functions implement a cryptographically secure pseudo-random number generator (PRNG). It is used by other library functions for example to generate random keys, and 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* (3). Its state can be saved in a seed file (see *RAND_load_file* (3)) to avoid having to go through the seeding process whenever the application is started.

RAND_bytes (3) describes how to obtain random data from the PRNG.

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:

First up I will state the things I believe I need for a good RNG.

- 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 being used to generate 4096 bit RSA keys, 2 2048 bit random strings are required (at a minimum). If your RNG state only has 128 bits, you are obviously limiting the search space to 128 bits, not 2048. I'm probably getting a little carried away on this last point but it does indicate that it may not be a bad idea to keep quite a lot of RNG state. 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 used should not be extractable from the RNG state. I 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', 2 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 chopped up into units of 20 bytes (or less for the last block). Each of these blocks 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 in 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. I believe this system addresses points 1 (hash function; currently SHA-1), 3 (the 'state'), 4 (via the 'md'), 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), we 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 we have finished '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').

So of the points raised, only 2 is not addressed (but see *RAND_add* (3)).

SEE ALSO

BN_rand (3), *RAND_add* (3), *RAND_load_file* (3), *RAND_egd* (3), *RAND_bytes* (3), *RAND_set_rand_method* (3), *RAND_cleanup* (3)

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

RAND_add() 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.

RAND_add() 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 RAND_add(), *RAND_egd* (3) or *RAND_load_file* (3).

RAND_seed() is equivalent to RAND_add() when *num* == *entropy*.

RAND_event() 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

RAND_status() and RAND_event() return 1 if the PRNG has been seeded with enough data, 0 otherwise.

The other functions do not return values.

SEE ALSO

rand (3), *RAND_egd* (3), *RAND_load_file* (3), *RAND_cleanup* (3)

HISTORY

RAND_seed() and RAND_screen() are available in all versions of SSLeay and OpenSSL. RAND_add() and RAND_status() have been added in OpenSSL 0.9.5, RAND_event() in OpenSSL 0.9.5a.

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

RAND_bytes() 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.

RAND_pseudo_bytes() puts *num* pseudo-random bytes into *buf*. Pseudo-random byte sequences generated by RAND_pseudo_bytes() 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

RAND_bytes() returns 1 on success, 0 otherwise. The error code can be obtained by *ERR_get_error* (3). RAND_pseudo_bytes() 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.

SEE ALSO

rand (3), *ERR_get_error* (3), *RAND_add* (3)

HISTORY

RAND_bytes() is available in all versions of SSLeay and OpenSSL. It has a return value since OpenSSL 0.9.5. RAND_pseudo_bytes() was added in OpenSSL 0.9.5.

RAND_cleanup

NAME

RAND_cleanup – erase the PRNG state

Synopsis

```
#include <openssl/rand.h>
void RAND_cleanup(void);
```

DESCRIPTION

RAND_cleanup() erases the memory used by the PRNG.

RETURN VALUE

RAND_cleanup() returns no value.

SEE ALSO

rand (3)

HISTORY

RAND_cleanup() is available in all versions of SSLeay and OpenSSL.

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);
int RAND_query_egd_bytes(const char *path, unsigned char *buf, int bytes);
```

DESCRIPTION

RAND_egd() queries the entropy gathering daemon EGD on socket *path*. It queries 255 bytes and uses *RAND_add* (3) to seed the OpenSSL built-in PRNG. RAND_egd(path) is a wrapper for RAND_egd_bytes(path, 255);

RAND_egd_bytes() queries the entropy gathering daemon EGD on socket *path*. It queries *bytes* bytes and uses *RAND_add* (3) to seed the OpenSSL built-in PRNG. This function is more flexible than RAND_egd(). 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.

RAND_query_egd_bytes() performs the actual query of the EGD daemon on socket *path*. If *buf* is given, *bytes* bytes are queried and written into *buf*. If *buf* is NULL, *bytes* bytes are queried and used to seed the OpenSSL built-in PRNG using *RAND_add* (3).

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 RAND_egd() is called with that path as an argument, it tries to read random bytes that EGD has 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.

OpenSSL automatically queries EGD when entropy is requested via RAND_bytes() or the status is checked via RAND_status() for the first time, if the socket is located at /var/run/egd-pool, /dev/egd-pool or /etc/egd-pool.

RETURN VALUE

RAND_egd() and RAND_egd_bytes() 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.

RAND_query_egd_bytes() returns the number of bytes read from the daemon on success, and -1 if the connection failed. The PRNG state is not considered.

SEE ALSO

rand (3), *RAND_add* (3), *RAND_cleanup* (3)

HISTORY

RAND_egd() is available since OpenSSL 0.9.5.

RAND_egd_bytes() is available since OpenSSL 0.9.6.

RAND_query_egd_bytes() is available since OpenSSL 0.9.7.

The automatic query of /var/run/egd-pool et al was added in OpenSSL 0.9.7.

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

RAND_file_name() 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.

RAND_load_file() reads a number of bytes from file *filename* and adds them to the PRNG. If *max_bytes* is non-negative, up to to *max_bytes* are read; starting with OpenSSL 0.9.5, if *max_bytes* is -1, the complete file is read.

RAND_write_file() 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.

RETURN VALUES

RAND_load_file() returns the number of bytes read.

RAND_write_file() returns the number of bytes written, and -1 if the bytes written were generated without appropriate seed.

RAND_file_name() returns a pointer to *buf* on success, and NULL on error.

SEE ALSO

rand (3), *RAND_add* (3), *RAND_cleanup* (3)

HISTORY

RAND_load_file(), RAND_write_file() and RAND_file_name() are available in all versions of SSLeay and OpenSSL.

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(const RAND_METHOD *meth);
const 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. IMPORTANT: See the NOTES section for important information about how these RAND API functions are affected by the use of *ENGINE* API calls.

Initially, the default RAND_METHOD is the OpenSSL internal implementation, as returned by RAND_SSLeay().

RAND_set_default_method() makes *meth* the method for PRNG use. *NB*: This is true only whilst no ENGINE has been set as a default for RAND, so this function is no longer recommended.

RAND_get_default_method() returns a pointer to the current RAND_METHOD. However, the meaningfulness of this result is dependant on whether the ENGINE API is being used, so this function is no longer recommended.

THE 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 RAND_seed(), RAND_bytes(), RAND_cleanup(), RAND_add(), RAND_pseudo_rand() and RAND_status(). Each component may be NULL if the function is not implemented.

RETURN VALUES

RAND_set_rand_method() returns no value. RAND_get_rand_method() and RAND_SSLeay() return pointers to the respective methods.

NOTES

As of version 0.9.7, RAND_METHOD implementations are grouped together with other algorithmic APIs (eg. RSA_METHOD, EVP_CIPHER, etc) in *ENGINE* modules. If a default ENGINE is specified for RAND functionality using an ENGINE API function, that will override any RAND defaults set using the RAND API (ie. RAND_set_rand_method()). For this reason, the ENGINE API is the recommended way to control default implementations for use in RAND and other cryptographic algorithms.

SEE ALSO

rand (3), *engine* (3)

HISTORY

RAND_set_rand_method(), RAND_get_rand_method() and RAND_SSLeay() are available in all versions of OpenSSL.

In the engine version of version 0.9.6, RAND_set_rand_method() was altered to take an ENGINE pointer as its argument. As of version 0.9.7, that has been reverted as the ENGINE API transparently overrides RAND defaults if used, otherwise RAND API functions work as before. RAND_set_rand_engine() was also introduced in version 0.9.7.

RC4_set_key

NAME

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 for example in *Applied Cryptography*. It is believed to be compatible with RC4[TM], 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.

RC4_set_key() sets up the *RC4_KEY key* using the *len* bytes long key at *data*.

RC4() 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 *EVP_EncryptInit* (3) etc. instead of calling the RC4 functions directly.

RETURN VALUES

RC4_set_key() and RC4() do not return values.

NOTE

Certain conditions have to be observed to securely use stream ciphers. It is not permissible to perform multiple encryptions using the same key stream.

SEE ALSO

blowfish (3), *des* (3), `rc2(3)`

HISTORY

RC4_set_key() and RC4() are available in all versions of SSLeay and OpenSSL.

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.

RIPEMD160() 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:

RIPEMD160_Init() initializes a *RIPEMD160_CTX* structure.

RIPEMD160_Update() can be called repeatedly with chunks of the message to be hashed (*len* bytes at *data*).

RIPEMD160_Final() 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 *EVP_DigestInit* (3) etc. instead of calling the hash functions directly.

RETURN VALUES

RIPEMD160() returns a pointer to the hash value.

RIPEMD160_Init(), RIPEMD160_Update() and RIPEMD160_Final() do not return values.

CONFORMING TO

ISO/IEC 10118-3 (draft) (??)

SEE ALSO

sha (3), *hmac* (3), *EVP_DigestInit* (3)

HISTORY

RIPEMD160(), RIPEMD160_Init(), RIPEMD160_Update() and RIPEMD160_Final() are available since SSLeay 0.9.0.

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_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(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);
int RSA_size(const 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_method(const RSA_METHOD *meth);
const RSA_METHOD *RSA_get_default_method(void);
int RSA_set_method(RSA *rsa, const RSA_METHOD *meth);
const RSA_METHOD *RSA_get_method(const RSA *rsa);
RSA_METHOD *RSA_PKCS1_SSLeay(void);
RSA_METHOD *RSA_null_method(void);
int RSA_flags(const 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_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*.

p, *q*, *dmp1*, *dmq1* and *iqmp* may be *NULL* in private keys, but the RSA operations are much faster when these values are available.

Note that RSA keys may use non-standard *RSA_METHOD* implementations, either directly or by the use of *ENGINE* modules. In some cases (eg. an ENGINE providing support for hardware-embedded keys), these BIGNUM values will not be used by the implementation or may be used for alternative data storage. For this reason, applications should generally avoid using RSA structure elements directly and instead use API functions to query or modify keys.

CONFORMING TO

SSL, PKCS #1 v2.0

PATENTS

RSA was covered by a US patent which expired in September 2000.

SEE ALSO

rsa (1), *bn* (3), *dsa* (3), *dh* (3), *rand* (3), *engine* (3), *RSA_new* (3), *RSA_public_encrypt* (3), *RSA_sign* (3), *RSA_size* (3), *RSA_generate_key* (3), *RSA_check_key* (3), *RSA_blinding_on* (3), *RSA_set_method* (3), *RSA_print* (3), *RSA_get_ex_new_index* (3), *RSA_private_encrypt* (3), RSA_sign_ASN1_*OCTET_STRING* (3), RSA_padding_add_PKCS1_type_1(3)

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.

RSA_blinding_on() turns blinding on for key *rsa* and generates a random blinding factor. *ctx* is *NULL* or a pre-allocated and initialized *BN_CTX*. The random number generator must be seeded prior to calling RSA_blinding_on().

RSA_blinding_off() turns blinding off and frees the memory used for the blinding factor.

RETURN VALUES

RSA_blinding_on() returns 1 on success, and 0 if an error occurred.

RSA_blinding_off() returns no value.

SEE ALSO

rsa (3), *rand* (3)

HISTORY

RSA_blinding_on() and RSA_blinding_off() appeared in SSLeay 0.9.0.

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 in fact 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*.

As such, this function can not be used with any arbitrary RSA key object, even if it is otherwise fit for regular RSA operation. See *NOTES* for more information.

RETURN VALUE

RSA_check_key() returns 1 if *rsa* is a valid RSA key, and 0 otherwise. -1 is returned if an error occurs while checking the key.

If the key is invalid or an error occurred, the reason code can be obtained using *ERR_get_error* (3).

NOTES

This function does not work on RSA public keys that have only the modulus and public exponent elements populated. It performs integrity checks on all the RSA key material, so the RSA key structure must contain all the private key data too.

Unlike most other RSA functions, this function does *not* work transparently with any underlying ENGINE implementation because it uses the key data in the RSA structure directly. An ENGINE implementation can override the way key data is stored and handled, and can even provide support for HSM keys - in which case the RSA structure may contain *no* key data at all! If the ENGINE in question is only being used for acceleration or analysis purposes, then in all likelihood the RSA key data is complete and untouched, but this can't be assumed in the general case.

Restrictions

A method of verifying the RSA key using opaque RSA API functions might need to be considered. Right now RSA_check_key() simply uses the RSA structure elements directly, bypassing the RSA_METHOD table altogether (and completely violating encapsulation and object-orientation in the process). The best fix will probably be to introduce a "check_key()" handler to the RSA_METHOD function table so that alternative implementations can also provide their own verifiers.

SEE ALSO

rsa (3), *ERR_get_error* (3)

HISTORY

RSA_check_key() appeared in OpenSSL 0.9.4.

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

RSA_generate_key() 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, 17 or 65537.

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* (3).
- 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)*.

RETURN VALUE

If key generation fails, RSA_generate_key() returns *NULL*; the error codes can be obtained by *ERR_get_error* (3).

Restrictions

callback(2, x, cb_arg) is used with two different meanings.

RSA_generate_key() goes into an infinite loop for illegal input values.

SEE ALSO

ERR_get_error (3), *rand* (3), *rsa* (3), *RSA_free* (3)

HISTORY

The *cb_arg* argument was added in SSLeay 0.9.0.

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 precise manner in which these function pointers are called is described in more detail below. *RSA_get_ex_new_index()* 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.

RSA_set_ex_data() is used to set application specific data, the data is supplied in the *arg* parameter and its precise meaning is up to the application.

RSA_get_ex_data() 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.

new_func() is called when a structure is initially allocated (for example with *RSA_new()*. 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.

free_func() 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.

new_func() and *free_func()* take the same parameters. *parent* is a pointer to the parent RSA structure. *ptr* is a the application specific data (this wont be of much use in *new_func()*. *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 *RSA_get_ex_new_index()* when the functions were initially registered. Finally the *argl* and *argp* parameters are the values originally passed to the same corresponding parameters when *RSA_get_ex_new_index()* was called.

dup_func() 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 *new_func()* and *free_func()*.

RETURN VALUES

RSA_get_ex_new_index() returns a new index or -1 on failure (note 0 is a valid index value).

RSA_set_ex_data() returns 1 on success or 0 on failure.

RSA_get_ex_data() 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.

new_func() and *dup_func()* should return 0 for failure and 1 for success.

On failure an error code can be obtained from *ERR_get_error* (3).

Restrictions

dup_func() is currently never called.

The return value of *new_func()* is ignored.

The *new_func()* function isn't very useful because no meaningful values are present in the parent RSA structure when it is called.

SEE ALSO

rsa (3), *CRYPTO_set_ex_data* (3)

HISTORY

RSA_get_ex_new_index(), RSA_set_ex_data() and RSA_get_ex_data() are available since SSLeay 0.9.0.

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

RSA_new() allocates and initializes an *RSA* structure. It is equivalent to calling RSA_new_method(NULL).

RSA_free() 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, RSA_new() returns *NULL* and sets an error code that can be obtained by *ERR_get_error* (3). Otherwise it returns a pointer to the newly allocated structure.

RSA_free() returns no value.

SEE ALSO

ERR_get_error (3), *rsa* (3), *RSA_generate_key* (3), *RSA_new_method* (3)

HISTORY

RSA_new() and RSA_free() are available in all versions of SSLeay and OpenSSL.

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. RSA_padding_add_PKCS1_OAEP() and RSA_padding_check_PKCS1_OAEP() may be used in an application combined with *RSA_NO_PADDING* in order to implement OAEP with an encoding parameter.

RSA_padding_add_xxx() encodes *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 RSA_padding_add_xxx().

RSA_padding_check_xxx() verifies that the *fl* bytes at *f* contain a valid encoding for a *rsa_len* byte RSA key in the respective encoding method and 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*. *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* (3).

SEE ALSO

RSA_public_encrypt (3), *RSA_private_decrypt* (3), *RSA_sign* (3), *RSA_verify* (3)

HISTORY

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() appeared in SSLeay 0.9.0.

RSA_padding_add_PKCS1_OAEP() and RSA_padding_check_PKCS1_OAEP() were added in OpenSSL 0.9.2b.

RSA_print

NAME

RSA_print, RSA_print_fp, DSAparams_print, DSAparams_print_fp, DSA_print, DSA_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.

SEE ALSO

dh (3), *dsa* (3), *rsa* (3), BN_bn2*bin* (3)

HISTORY

RSA_print(), RSA_print_fp(), DSA_print(), DSA_print_fp(), DH_print(), DH_print_fp() are available in all versions of SSLeay and OpenSSL. DSAparams_print() and DSAparams_print_pf() were added in SSLeay 0.8.

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.

RSA_private_encrypt() 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*. *to* must point to *RSA_size(rsa)* bytes of memory.

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, *RSA_sign* (3) and *RSA_verify* (3) 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.

RSA_public_decrypt() recovers the message digest from the *flen* bytes long signature at *from* using the signer's public key *rsa*. *to* must point to a memory section large enough to hold the message digest (which is smaller than *RSA_size(rsa) - 11*). *padding* is the padding mode that was used to sign the data.

RETURN VALUES

RSA_private_encrypt() returns the size of the signature (i.e., RSA_size(rsa)). RSA_public_decrypt() returns the size of the recovered message digest.

On error, -1 is returned; the error codes can be obtained by *ERR_get_error* (3).

SEE ALSO

ERR_get_error (3), *rsa* (3), *RSA_sign* (3), *RSA_verify* (3)

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

RSA_public_encrypt() encrypts the *flen* bytes at *from* (usually a session key) using the public key *rsa* and stores the ciphertext in *to*. *to* must point to RSA_size(*rsa*) bytes of memory.

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.

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 RSA_public_encrypt().

RSA_private_decrypt() decrypts the *flen* bytes at *from* using the private key *rsa* and stores the plaintext in *to*. *to* must point to a memory section large enough to hold the decrypted data (which is smaller than RSA_size(*rsa*)). *padding* is the padding mode that was used to encrypt the data.

RETURN VALUES

RSA_public_encrypt() returns the size of the encrypted data (i.e., RSA_size(*rsa*)). RSA_private_decrypt() returns the size of the recovered plaintext.

On error, -1 is returned; the error codes can be obtained by *ERR_get_error* (3).

CONFORMING TO

SSL, PKCS #1 v2.0

SEE ALSO

ERR_get_error (3), *rand* (3), *rsa* (3), *RSA_size* (3)

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.

RSA_set_default_method

NAME

RSA_set_default_method, RSA_get_default_method, RSA_set_method, RSA_get_method, RSA_PKCS1_SSLeay, RSA_null_method, RSA_flags, RSA_new_method – select RSA method

Synopsis

```
#include <openssl/rsa.h>
void RSA_set_default_method(const RSA_METHOD *meth);
RSA_METHOD *RSA_get_default_method(void);
int RSA_set_method(RSA *rsa, const RSA_METHOD *meth);
RSA_METHOD *RSA_get_method(const RSA *rsa);
RSA_METHOD *RSA_PKCS1_SSLeay(void);
RSA_METHOD *RSA_null_method(void);
int RSA_flags(const RSA *rsa);
RSA *RSA_new_method(RSA_METHOD *method);
```

DESCRIPTION

An *RSA_METHOD* specifies the functions that OpenSSL uses for RSA operations. By modifying the method, alternative implementations such as hardware accelerators may be used. IMPORTANT: See the NOTES section for important information about how these RSA API functions are affected by the use of *ENGINE* API calls.

Initially, the default RSA_METHOD is the OpenSSL internal implementation, as returned by RSA_PKCS1_SSLeay().

RSA_set_default_method() makes *meth* the default method for all RSA structures created later. *NB*: This is true only whilst no ENGINE has been set as a default for RSA, so this function is no longer recommended.

RSA_get_default_method() returns a pointer to the current default RSA_METHOD. However, the meaningfulness of this result is dependant on whether the ENGINE API is being used, so this function is no longer recommended.

RSA_set_method() selects *meth* to perform all operations using the key *rsa*. This will replace the RSA_METHOD used by the RSA key and if the previous method was supplied by an ENGINE, the handle to that ENGINE will be released during the change. It is possible to have RSA keys that only work with certain RSA_METHOD implementations (eg. from an ENGINE module that supports embedded hardware-protected keys), and in such cases attempting to change the RSA_METHOD for the key can have unexpected results.

RSA_get_method() returns a pointer to the RSA_METHOD being used by *rsa*. This method may or may not be supplied by an ENGINE implementation, but if it is, the return value can only be guaranteed to be valid as long as the RSA key itself is valid and does not have its implementation changed by RSA_set_method().

RSA_flags() returns the *flags* that are set for *rsa*'s current RSA_METHOD. See the Restrictions section.

RSA_new_method() 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, and if no default ENGINE is set, the RSA_METHOD controlled by RSA_set_default_method() is used.

RSA_flags() returns the *flags* that are set for *rsa*'s current method.

RSA_new_method() allocates and initializes an *RSA* structure so that *method* will be used for the RSA operations. If *method* is *NULL*, the default method is used.

THE 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

RSA_PKCS1_SSLeay(), RSA_PKCS1_null_method(), RSA_get_default_method() and RSA_get_method() return pointers to the respective RSA_METHODs.

RSA_set_default_method() returns no value.

RSA_set_method() returns a pointer to the old RSA_METHOD implementation that was replaced. However, this return value should probably be ignored because if it was supplied by an ENGINE, the pointer could be invalidated at any time if the ENGINE is unloaded (in fact it could be unloaded as a result of the RSA_set_method() function releasing its handle to the ENGINE). For this reason, the return type may be replaced with a *void* declaration in a future release.

RSA_new_method() returns NULL and sets an error code that can be obtained by *ERR_get_error* (3) if the allocation fails. Otherwise it returns a pointer to the newly allocated structure.

NOTES

As of version 0.9.7, RSA_METHOD implementations are grouped together with other algorithmic APIs (eg. DSA_METHOD, EVP_CIPHER, etc) into *ENGINE* modules. If a default ENGINE is specified for RSA functionality using an ENGINE API function, that will override any RSA defaults set using the RSA API (ie. RSA_set_default_method()). For this reason, the ENGINE API is the recommended way to control default implementations for use in RSA and other cryptographic algorithms.

Restrictions

The behaviour of RSA_flags() is a mis-feature that is left as-is for now to avoid creating compatibility problems. RSA functionality, such as the encryption functions, are controlled by the *flags* value in the RSA key itself, not by the *flags* value in the RSA_METHOD attached to the RSA key (which is what this function returns). If the flags element of an RSA key is changed, the changes will be honoured by RSA functionality but will not be reflected in the return value of the RSA_flags() function - in effect RSA_flags() behaves more like an RSA_default_flags() function (which does not currently exist).

SEE ALSO

rsa (3), *RSA_new* (3)

HISTORY

RSA_new_method() and RSA_set_default_method() appeared in SSLeay 0.8. RSA_get_default_method(), RSA_set_method() and RSA_get_method() as well as the rsa_sign and rsa_verify components of RSA_METHOD were added in OpenSSL 0.9.4.

RSA_set_default_openssl_method() and RSA_get_default_openssl_method() replaced RSA_set_default_method() and RSA_get_default_method() respectively, and RSA_set_method() and RSA_new_method() were altered to use *ENGINE*s rather than *RSA_METHOD*s during development of the engine version of OpenSSL 0.9.6. For 0.9.7, the handling of defaults in the ENGINE API was restructured so that this change was reversed, and behaviour of the other functions resembled more closely the previous behaviour. The behaviour of defaults in the ENGINE API now transparently overrides the behaviour of defaults in the RSA API without requiring changing these function prototypes.

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

RSA_sign() 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*. *sigret* must point to RSA_size(*rsa*) bytes of memory.

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* (3) 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.

RSA_verify() verifies that the signature *sigbuf* of size *siglen* matches a given message digest *m* of size *m_len*. *type* denotes the message digest algorithm that was used to generate the signature. *rsa* is the signer's public key.

RETURN VALUES

RSA_sign() returns 1 on success, 0 otherwise. RSA_verify() returns 1 on successful verification, 0 otherwise.

The error codes can be obtained by *ERR_get_error* (3).

Restrictions

Certain signatures with an improper algorithm identifier are accepted for compatibility with SSLeay 0.4.5 :-)

CONFORMING TO

SSL, PKCS #1 v2.0

SEE ALSO

ERR_get_error (3), *objects* (3), *rsa* (3), *RSA_private_encrypt* (3), *RSA_public_decrypt* (3)

HISTORY

RSA_sign() and RSA_verify() are available in all versions of SSLeay and OpenSSL.

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

RSA_sign_ASN1_OCTET_STRING() 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*. *sigret* must point to *RSA_size(rsa)* bytes of memory.

dummy is ignored.

The random number generator must be seeded prior to calling RSA_sign_ASN1_OCTET_STRING().

RSA_verify_ASN1_OCTET_STRING() verifies that the signature *sigbuf* of size *siglen* is the DER representation of a given octet string *m* of size *m_len*. *dummy* is ignored. *rsa* is the signer's public key.

RETURN VALUES

RSA_sign_ASN1_OCTET_STRING() returns 1 on success, 0 otherwise.
RSA_verify_ASN1_OCTET_STRING() returns 1 on successful verification, 0 otherwise.

The error codes can be obtained by *ERR_get_error* (3).

Restrictions

These functions serve no recognizable purpose.

SEE ALSO

ERR_get_error (3), *objects* (3), *rand* (3), *rsa* (3), *RSA_sign* (3), *RSA_verify* (3)

HISTORY

RSA_sign_ASN1_OCTET_STRING() and RSA_verify_ASN1_OCTET_STRING() were added in SSLeay 0.8.

RSA_size

NAME

RSA_size – get RSA modulus size

Synopsis

```
#include <openssl/rsa.h>
int RSA_size(const 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.

rsa->n must not be *NULL*.

RETURN VALUE

The size in bytes.

SEE ALSO

rsa (3)

HISTORY

RSA_size() is available in all versions of SSLeay and OpenSSL.

SHA1

NAME

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.

SHA1() 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 may be used if the message is not completely stored in memory:

SHA1_Init() initializes a *SHA_CTX* structure.

SHA1_Update() can be called repeatedly with chunks of the message to be hashed (*len* bytes at *data*).

SHA1_Final() 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 *EVP_DigestInit* (3) etc. 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

SHA1() returns a pointer to the hash value.

SHA1_Init(), SHA1_Update() and SHA1_Final() do not return values.

CONFORMING TO

SHA: US Federal Information Processing Standard FIPS PUB 180 (Secure Hash Standard), SHA-1: US Federal Information Processing Standard FIPS PUB 180-1 (Secure Hash Standard), ANSI X9.30

SEE ALSO

ripemd (3), *hmac* (3), *EVP_DigestInit* (3)

HISTORY

SHA1(), SHA1_Init(), SHA1_Update() and SHA1_Final() are available in all versions of SSLeay and OpenSSL.

SMIME_read_PKCS7

NAME

SMIME_read_PKCS7 – parse S/MIME message.

Synopsis

```
PKCS7 *SMIME_read_PKCS7(BIO *in, BIO **bcont);
```

DESCRIPTION

SMIME_read_PKCS7() parses a message in S/MIME format.

in is a BIO to read the message from.

If cleartext signing is used then the content is saved in a memory bio which is written to **bcont*, otherwise **bcont* is set to *NULL*.

The parsed PKCS#7 structure is returned or *NULL* if an error occurred.

NOTES

If **bcont* is not *NULL* then the message is clear text signed. **bcont* can then be passed to PKCS7_verify() with the *PKCS7_DETACHED* flag set.

Otherwise the type of the returned structure can be determined using PKCS7_type().

To support future functionality if *bcont* is not *NULL* **bcont* should be initialized to *NULL*. For example:

```
BIO *cont = NULL;
PKCS7 *p7;

p7 = SMIME_read_PKCS7(in, &cont);
```

Restrictions

The MIME parser used by SMIME_read_PKCS7() is somewhat primitive. While it will handle most S/MIME messages more complex compound formats may not work.

The parser assumes that the PKCS7 structure is always base64 encoded and will not handle the case where it is in binary format or uses quoted printable format.

The use of a memory BIO to hold the signed content limits the size of message which can be processed due to memory restraints: a streaming single pass option should be available.

RETURN VALUES

SMIME_read_PKCS7() returns a valid *PKCS7* structure or *NULL* is an error occurred. The error can be obtained from *ERR_get_error* (3).

SEE ALSO

ERR_get_error (3), PKCS7_*type* (3) `SMIME_read_PKCS7(3)`, PKCS7_*sign* (3), PKCS7_*verify* (3), PKCS7_*encrypt* (3) PKCS7_*decrypt* (3)

HISTORY

SMIME_read_PKCS7() was added to OpenSSL 0.9.5

SMIME_write_PKCS7

NAME

SMIME_write_PKCS7 – convert PKCS#7 structure to S/MIME format.

Synopsis

```
int SMIME_write_PKCS7(BIO *out, PKCS7 *p7, BIO *data, int flags);
```

DESCRIPTION

SMIME_write_PKCS7() adds the appropriate MIME headers to a PKCS#7 structure to produce an S/MIME message.

out is the BIO to write the data to. *p7* is the appropriate *PKCS7* structure. If cleartext signing (*multipart/signed*) is being used then the signed data must be supplied in the *data* argument. *flags* is an optional set of flags.

NOTES

The following flags can be passed in the *flags* parameter.

If *PKCS7_DETACHED* is set then cleartext signing will be used, this option only makes sense for signedData where *PKCS7_DETACHED* is also set when PKCS7_sign() is also called.

If the *PKCS7_TEXT* flag is set MIME headers for type *text/plain* are added to the content, this only makes sense if *PKCS7_DETACHED* is also set.

If cleartext signing is being used then the data must be read twice: once to compute the signature in PKCS7_sign() and once to output the S/MIME message.

Restrictions

SMIME_write_PKCS7() always base64 encodes PKCS#7 structures, there should be an option to disable this.

There should really be a way to produce cleartext signing using only a single pass of the data.

RETURN VALUES

SMIME_write_PKCS7() returns 1 for success or 0 for failure.

SEE ALSO

ERR_get_error (3), PKCS7_*sign* (3), PKCS7_*verify* (3), PKCS7_*encrypt* (3) PKCS7_*decrypt* (3)

HISTORY

SMIME_write_PKCS7() was added to OpenSSL 0.9.5

CRYPTO_set_locking_callback

NAME

CRYPTO_set_locking_callback, CRYPTO_set_id_callback, CRYPTO_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_function)(int mode, int n, const char
*file, int line));
void CRYPTO_set_id_callback(unsigned long (*id_function)(void)); 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_function) (int 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.

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.

locking_function() must be able to handle up to CRYPTO_num_locks() different mutex locks. It sets the *n*-th lock if *mode* & *CRYPTO_LOCK*, and releases it otherwise.

file and *line* are the file number of the function setting the lock. They can be useful for debugging.

id_function(void) is a function that 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 sometimes, some parts of OpenSSL need it for better performance. To enable this, the following is required:

- Three additional callback function, dyn_create_function, dyn_lock_function and dyn_destroy_function.
- A structure defined with the data that each lock needs to handle.

struct CRYPTO_dynlock_value has to be defined to contain whatever structure is needed to handle locks.

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.

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.

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.

CRYPTO_get_new_dynlockid() is used to create locks. It will call dyn_create_function for the actual creation.

CRYPTO_destroy_dynlockid() is used to destroy locks. It will call dyn_destroy_function for the actual destruction.

CRYPTO_lock() is used to lock and unlock the locks. mode is a bitfield describing what should be done with the lock. n is the number of the lock as returned from CRYPTO_get_new_dynlockid(). mode can be combined from the following values. These values are pairwise exclusive, with undefined behaviour if misused (for example, CRYPTO_READ and CRYPTO_WRITE should not be used together):

```
CRYPTO_LOCK0x01
CRYPTO_UNLOCK0x02
CRYPTO_READ0x04
CRYPTO_WRITE0x08
```

RETURN VALUES

CRYPTO_num_locks() returns the required number of locks.

CRYPTO_get_new_dynlockid() returns the index to the newly created lock.

The other functions return no values.

NOTE

You can find out 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 currently not used internally by OpenSSL, but may do so in the future.

EXAMPLES

crypto/threads/mttest.c shows examples of the callback functions on Solaris, Irix and Win32.

HISTORY

CRYPTO_set_locking_callback() and CRYPTO_set_id_callback() are available in all versions of SSLeay and OpenSSL. CRYPTO_num_locks() was added in OpenSSL 0.9.4. All functions dealing with dynamic locks were added in OpenSSL 0.9.5b-dev.

SEE ALSO

crypto (3)

UI_new

NAME

UI_new, UI_new_method, UI_free, UI_add_input_string, UI_dup_input_string, UI_add_verify_string, UI_dup_verify_string, UI_add_input_boolean, UI_dup_input_boolean, UI_add_info_string, UI_dup_info_string, UI_add_error_string, UI_dup_error_string, UI_construct_prompt, UI_add_user_data, UI_get0_user_data, UI_get0_result, UI_process, UI_ctrl, UI_set_default_method, UI_get_default_method, UI_get_method, UI_set_method, UI_OpenSSL, ERR_load_UI_strings – New User Interface

Synopsis

```
#include <openssl/ui.h>
typedef struct ui_st UI;
typedef struct ui_method_st UI_METHOD;
UI *UI_new(void);
UI *UI_new_method(const UI_METHOD *method);
void UI_free(UI *ui);
int UI_add_input_string(UI *ui, const char *prompt, int flags, char *result_buf, int
minsize, int maxsize);
int UI_dup_input_string(UI *ui, const char *prompt, int flags, char *result_buf, int
minsize, int maxsize);
int UI_add_verify_string(UI *ui, const char *prompt, int flags, char *result_buf, int
minsize, int maxsize, const char *test_buf);
int UI_dup_verify_string(UI *ui, const char *prompt, int flags, char *result_buf, int
minsize, int maxsize, const char *test_buf);
int UI_add_input_boolean(UI *ui, const char *prompt, const char *action_desc, const char
*ok_chars, const char *cancel_chars, int flags, char *result_buf);
int UI_dup_input_boolean(UI *ui, const char *prompt, const char *action_desc, const char
*ok_chars, const char *cancel_chars, int flags, char *result_buf);
int UI_add_info_string(UI *ui, const char *text);
int UI_dup_info_string(UI *ui, const char *text);
int UI_add_error_string(UI *ui, const char *text);
int UI_dup_error_string(UI *ui, const char *text);
/* These are the possible flags. They can be or'ed together. */
#define UI_INPUT_FLAG_ECHO0x01
#define UI_INPUT_FLAG_DEFAULT_PWD0x02
char *UI_construct_prompt(UI *ui_method, const char *object_desc, const char
*object_name);
void *UI_add_user_data(UI *ui, void *user_data);
void *UI_get0_user_data(UI *ui);
const char *UI_get0_result(UI *ui, int i);
int UI_process(UI *ui);
int UI_ctrl(UI *ui, int cmd, long i, void *p, void (*f)());
#define UI_CTRL_PRINT_ERRORS1
#define UI_CTRL_IS_REDOABLE2
void UI_set_default_method(const UI_METHOD *meth);
const UI_METHOD *UI_get_default_method(void);
```

```
const UI_METHOD *UI_get_method(UI *ui);
const UI_METHOD *UI_set_method(UI *ui, const UI_METHOD *meth);
UI_METHOD *UI_OpenSSL(void);
```

DESCRIPTION

UI stands for User Interface, and is general purpose set of routines to prompt the user for text-based information. Through user-written methods (see *ui_create* (3)), prompting can be done in any way imaginable, be it plain text prompting, through dialog boxes or from a cell phone.

All the functions work through a context of the type UI. This context contains all the information needed to prompt correctly as well as a reference to a UI_METHOD, which is an ordered vector of functions that carry out the actual prompting.

The first thing to do is to create a UI with UI_new() or UI_new_method(), then add information to it with the UI_add or UI_dup functions. Also, user-defined random data can be passed down to the underlying method through calls to UI_add_user_data. The default UI method doesn't care about these data, but other methods might. Finally, use UI_process() to actually perform the prompting and UI_get0_result() to find the result to the prompt.

A UI can contain more than one prompt, which are performed in the given sequence. Each prompt gets an index number which is returned by the UI_add and UI_dup functions, and has to be used to get the corresponding result with UI_get0_result().

The functions are as follows:

UI_new() creates a new UI using the default UI method. When done with this UI, it should be freed using UI_free().

UI_new_method() creates a new UI using the given UI method. When done with this UI, it should be freed using UI_free().

UI_OpenSSL() returns the built-in UI method (note: not the default one, since the default can be changed. See further on). This method is the most machine/OS dependent part of OpenSSL and normally generates the most problems when porting.

UI_free() removes a UI from memory, along with all other pieces of memory that's connected to it, like duplicated input strings, results and others.

UI_add_input_string() and UI_add_verify_string() add a prompt to the UI, as well as flags and a result buffer and the desired minimum and maximum sizes of the result. The given information is used to prompt for information, for example a password, and to verify a password (i.e. having the user enter it twice and check that the same string was entered twice). UI_add_verify_string() takes and extra argument that should be a pointer to the result buffer of the input string that it's supposed to verify, or verification will fail.

UI_add_input_boolean() adds a prompt to the UI that's supposed to be answered in a boolean way, with a single character for yes and a different character for no. A set of characters that can be used to cancel the prompt is given as well. The prompt itself is really divided in two, one part being the descriptive text (given through the *prompt* argument) and one describing the possible answers (given through the *action_desc* argument).

UI_add_info_string() and UI_add_error_string() add strings that are shown at the same time as the prompt for extra information or to show an error string. The difference between the two is only conceptual. With the builtin method, there's no technical difference between them. Other methods may make a difference between them, however.

The flags currently supported are UI_INPUT_FLAG_ECHO, which is relevant for UI_add_input_string() and will have the users response be echoed (when prompting for a password, this flag should obviously not be used, and UI_INPUT_FLAG_DEFAULT_PWD, which means that a default password of some sort will be used (completely depending on the application and the UI method).

UI_dup_input_string(), UI_dup_verify_string(), UI_dup_input_boolean(), UI_dup_info_string() and UI_dup_error_string() are basically the same as their UI_add counterparts, except that they make their own copies of all strings.

UI_construct_prompt() is a helper function that can be used to create a prompt from two pieces of information: an description and a name. The default constructor (if there is none provided by the method used) creates a string "Enter *description* for *name*:". With the description "pass phrase" and the file name "foo.key", that becomes "Enter pass phrase for foo.key:". Other methods may create whatever string and may include encodings that will be processed by the other method functions.

UI_add_user_data() adds a piece of memory for the method to use at any time. The builtin UI method doesn't care about this info. Note that several calls to this function doesn't add data, it replaces the previous blob with the one given as argument.

UI_get0_user_data() retrieves the data that has last been given to the UI with UI_add_user_data().

UI_get0_result() returns a pointer to the result buffer associated with the information indexed by *i*.

UI_process() goes through the information given so far, does all the printing and prompting and returns.

UI_ctrl() adds extra control for the application author. For now, it understands two commands: UI_CTRL_PRINT_ERRORS, which makes UI_process() print the OpenSSL error stack as part of processing the UI, and UI_CTRL_IS_REDOABLE, which returns a flag saying if the used UI can be used again or not.

UI_set_default_method() changes the default UI method to the one given.

UI_get_default_method() returns a pointer to the current default UI method.

UI_get_method() returns the UI method associated with a given UI.

UI_set_method() changes the UI method associated with a given UI.

SEE ALSO

ui_create (3), *ui_compat* (3)

HISTORY

The UI section was first introduced in OpenSSL 0.9.7.

AUTHOR

Richard Levitte (richard@levitte.org) for the OpenSSL project (http://www.openssl.org).

des_read_password

NAME

des_read_password, des_read_2passwords, des_read_pw_string, des_read_pw – Compatibility user interface functions

Synopsis

```
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);
int des_read_pw(char *buf,char *buff,int size,const char *prompt,int verify);
```

DESCRIPTION

The DES library contained a few routines to prompt for passwords. These aren't necessarely dependent on DES, and have therefore become part of the UI compatibility library.

des_read_pw() writes the string specified by *prompt* to standard output turns echo off and reads an input string from the terminal. The string is returned in *buf*, which must have spac for at least *size* bytes. If *verify* is set, the user is asked for the password twice and unless the two copies match, an error is returned. The second password is stored in *buff*, which must therefore also be at least *size* bytes. A return code of -1 indicates a system error, 1 failure due to use interaction, and 0 is success. All other functions described here use des_read_pw() to do the work.

des_read_pw_string() is a variant of des_read_pw() that provides a buffer for you if *verify* is set.

des_read_password() calls des_read_pw() and converts the password to a DES key by calling DES_string_to_key(); des_read_2password() operates in the same way as des_read_password() except that it generates two keys by using the DES_string_to_2key() function.

NOTES

des_read_pw_string() is available in the MIT Kerberos library as well, and is also available under the name EVP_read_pw_string().

SEE ALSO

ui (3), *ui_create* (3)

AUTHOR

Richard Levitte (richard@levitte.org) for the OpenSSL project (http://www.openssl.org).

X509_NAME_add_entry_by_txt

NAME

X509_NAME_add_entry_by_txt, X509_NAME_add_entry_by_OBJ,

X509_NAME_add_entry_by_NID, X509_NAME_add_entry, X509_NAME_delete_entry – X509_NAME modification functions

Synopsis

```
int X509_NAME_add_entry_by_txt(X509_NAME *name, char *field, int type, unsigned char
*bytes, int len, int loc, int set);
int X509_NAME_add_entry_by_OBJ(X509_NAME *name, ASN1_OBJECT *obj, int type, unsigned char
*bytes, int len, int loc, int set);
int X509_NAME_add_entry_by_NID(X509_NAME *name, int nid, int type, unsigned char *bytes,
int len, int loc, int set);
int X509_NAME_add_entry(X509_NAME *name,X509_NAME_ENTRY *ne, int loc, int set);
X509_NAME_ENTRY *X509_NAME_delete_entry(X509_NAME *name, int loc);
```

DESCRIPTION

X509_NAME_add_entry_by_txt(), X509_NAME_add_entry_by_OBJ() and X509_NAME_add_entry_by_NID() add a field whose name is defined by a string *field*, an object *obj* or a NID *nid* respectively. The field value to be added is in *bytes* of length *len*. If *len* is -1 then the field length is calculated internally using strlen(bytes).

The type of field is determined by *type* which can either be a definition of the type of *bytes* (such as *MBSTRING_ASC*) or a standard ASN1 type (such as *V_ASN1_IA5STRING*). The new entry is added to a position determined by *loc* and *set*.

X509_NAME_add_entry() adds a copy of *X509_NAME_ENTRY* structure *ne* to *name*. The new entry is added to a position determined by *loc* and *set*. Since a copy of *ne* is added *ne* must be freed up after the call.

X509_NAME_delete_entry() deletes an entry from *name* at position *loc*. The deleted entry is returned and must be freed up.

NOTES

The use of string types such as *MBSTRING_ASC* or *MBSTRING_UTF8* is strongly recommened for the *type* parameter. This allows the internal code to correctly determine the type of the field and to apply length checks according to the relevant standards. This is done using ASN1_STRING_set_by_NID().

If instead an ASN1 type is used no checks are performed and the supplied data in *bytes* is used directly.

In X509_NAME_add_entry_by_txt() the *field* string represents the field name using OBJ_txt2obj(field, 0).

The *loc* and *set* parameters determine where a new entry should be added. For almost all applications *loc* can be set to -1 and *set* to 0. This adds a new entry to the end of *name* as a single valued RelativeDistinguishedName (RDN).

loc actually determines the index where the new entry is inserted: if it is -1 it is appended.

set determines how the new type is added. If it is zero a new RDN is created.

If *set* is -1 or 1 it is added to the previous or next RDN structure respectively. This will then be a multivalued RDN: since multivalues RDNs are very seldom used *set* is almost always set to zero.

EXAMPLES

Create an *X509_NAME* structure: "C=UK, O=Disorganized Organization, CN=Joe Bloggs"

```
 X509_NAME *nm;
 nm = X509_NAME_new();
 if (nm == NULL)
/* Some error */
 if (!X509_NAME_add_entry_by_txt(nm, MBSTRING_ASC,
"C", "UK", -1, -1, 0))
/* Error */
 if (!X509_NAME_add_entry_by_txt(nm, MBSTRING_ASC,
"O", "Disorganized Organization", -1, -1, 0))
/* Error */
 if (!X509_NAME_add_entry_by_txt(nm, MBSTRING_ASC,
"CN", "Joe Bloggs", -1, -1, 0))
/* Error */
```

RETURN VALUES

X509_NAME_add_entry_by_txt(), X509_NAME_add_entry_by_OBJ(), X509_NAME_add_entry_by_NID() and X509_NAME_add_entry() return 1 for success of 0 if an error occurred.

X509_NAME_delete_entry() returns either the deleted *X509_NAME_ENTRY* structure of *NULL* if an error occurred.

Restrictions

type can still be set to *V_ASN1_APP_CHOOSE* to use a different algorithm to determine field types. Since this form does not understand multicharacter types, performs no length checks and can result in invalid field types its use is strongly discouraged.

SEE ALSO

ERR_get_error (3), d2i_X509_*NAME* (3)

HISTORY

None.

X509_NAME_ENTRY_get_object

NAME

X509_NAME_ENTRY_get_object, X509_NAME_ENTRY_get_data,

X509_NAME_ENTRY_set_object, X509_NAME_ENTRY_set_data,

X509_NAME_ENTRY_create_by_txt, X509_NAME_ENTRY_create_by_NID,

X509_NAME_ENTRY_create_by_OBJ – X509_NAME_ENTRY utility functions

Synopsis

```
ASN1_OBJECT * X509_NAME_ENTRY_get_object(X509_NAME_ENTRY *ne);
ASN1_STRING * X509_NAME_ENTRY_get_data(X509_NAME_ENTRY *ne);
int X509_NAME_ENTRY_set_object(X509_NAME_ENTRY *ne, ASN1_OBJECT *obj);
int X509_NAME_ENTRY_set_data(X509_NAME_ENTRY *ne, int type, unsigned char *bytes, int
len);
X509_NAME_ENTRY *X509_NAME_ENTRY_create_by_txt(X509_NAME_ENTRY **ne, char *field, int
type, unsigned char *bytes, int len);
X509_NAME_ENTRY *X509_NAME_ENTRY_create_by_NID(X509_NAME_ENTRY **ne, int nid, int
type,unsigned char *bytes, int len);
X509_NAME_ENTRY *X509_NAME_ENTRY_create_by_OBJ(X509_NAME_ENTRY **ne, ASN1_OBJECT *obj, int
type,unsigned char *bytes, int len);
```

DESCRIPTION

X509_NAME_ENTRY_get_object() retrieves the field name of *ne* in and *ASN1_OBJECT* structure.

X509_NAME_ENTRY_get_data() retrieves the field value of *ne* in and *ASN1_STRING* structure.

X509_NAME_ENTRY_set_object() sets the field name of *ne* to *obj*.

X509_NAME_ENTRY_set_data() sets the field value of *ne* to string type *type* and value determined by *bytes* and *len*.

X509_NAME_ENTRY_create_by_txt(), X509_NAME_ENTRY_create_by_NID() and X509_NAME_ENTRY_create_by_OBJ() create and return an *X509_NAME_ENTRY* structure.

NOTES

X509_NAME_ENTRY_get_object() and X509_NAME_ENTRY_get_data() can be used to examine an *X509_NAME_ENTRY* function as returned by X509_NAME_get_entry() for example.

X509_NAME_ENTRY_create_by_txt(), X509_NAME_ENTRY_create_by_NID(), and X509_NAME_ENTRY_create_by_OBJ() create and return an

X509_NAME_ENTRY_create_by_txt(), X509_NAME_ENTRY_create_by_OBJ(), X509_NAME_ENTRY_create_by_NID() and X509_NAME_ENTRY_set_data() are seldom used in practice because *X509_NAME_ENTRY* structures are almost always part of *X509_NAME* structures and the corresponding *X509_NAME* functions are typically used to create and add new entries in a single operation.

The arguments of these functions support similar options to the similarly named ones of the corresponding *X509_NAME* functions such as X509_NAME_add_entry_by_txt(). So for example *type* can be set to *MBSTRING_ASC* but in the case of X509_set_data() the field name must be set first so the relevant field information can be looked up internally.

RETURN VALUES

None.

SEE ALSO

ERR_get_error (3), d2i_X509_*NAME* (3), OBJ_nid2*obj* (3),OBJ_nid2*obj* (3)

HISTORY

None.

X509_NAME_get_index_by_NID

NAME

X509_NAME_get_index_by_NID, X509_NAME_get_index_by_OBJ, X509_NAME_get_entry,

X509_NAME_entry_count, X509_NAME_get_text_by_NID, X509_NAME_get_text_by_OBJ – X509_NAME lookup and enumeration functions

Synopsis

```
int X509_NAME_get_index_by_NID(X509_NAME *name,int nid,int lastpos);
int X509_NAME_get_index_by_OBJ(X509_NAME *name,ASN1_OBJECT *obj, int lastpos);
int X509_NAME_entry_count(X509_NAME *name);
X509_NAME_ENTRY *X509_NAME_get_entry(X509_NAME *name, int loc);
int X509_NAME_get_text_by_NID(X509_NAME *name, int nid, char *buf,int len);
int X509_NAME_get_text_by_OBJ(X509_NAME *name, ASN1_OBJECT *obj, char *buf,int len);
```

DESCRIPTION

These functions allow an *X509_NAME* structure to be examined. The *X509_NAME* structure is the same as the *Name* type defined in RFC2459 (and elsewhere) and used for example in certificate subject and issuer names.

X509_NAME_get_index_by_NID() and X509_NAME_get_index_by_OBJ() retrieve the next index matching *nid* or *obj* after *lastpos*. *lastpos* should initially be set to -1. If there are no more entries -1 is returned.

X509_NAME_entry_count() returns the total number of entries in *name*.

X509_NAME_get_entry() retrieves the *X509_NAME_ENTRY* from *name* corresponding to index *loc*. Acceptable values for *loc* run from 0 to (X509_NAME_entry_count(name) - 1). The value returned is an internal pointer which must not be freed.

X509_NAME_get_text_by_NID(), X509_NAME_get_text_by_OBJ() retrieve the "text" from the first entry in *name* which matches *nid* or *obj*, if no such entry exists -1 is returned. At most *len* bytes will be written and the text written to *buf* will be null terminated. The length of the output string written is returned excluding the terminating null. If *buf* is <NULL> then the amount of space needed in *buf* (excluding the final null) is returned.

NOTES

X509_NAME_get_text_by_NID() and X509_NAME_get_text_by_OBJ() are legacy functions which have various limitations which make them of minimal use in practice. They can only find the first matching entry and will copy the contents of the field verbatim: this can be highly confusing if the target is a muticharacter string type like a BMPString or a UTF8String.

For a more general solution X509_NAME_get_index_by_NID() or X509_NAME_get_index_by_OBJ() should be used followed by X509_NAME_get_entry() on any matching indices and then the various *X509_NAME_ENTRY* utility functions on the result.

EXAMPLES

Process all entries:

```
int i;
X509_NAME_ENTRY *e;
```

```
 for (i = 0; i < X509_NAME_entry_count(nm); i++)
{
e = X509_NAME_get_entry(nm, i);
/* Do something with e */
}
```

Process all commonName entries:

```
 int loc;
 X509_NAME_ENTRY *e;

 loc = -1;
 for (;;)
{
lastpos = X509_NAME_get_index_by_NID(nm, NID_commonName, lastpos);
if (lastpos == -1)
break;
e = X509_NAME_get_entry(nm, lastpos);
/* Do something with e */
}
```

RETURN VALUES

X509_NAME_get_index_by_NID() and X509_NAME_get_index_by_OBJ() return the index of the next matching entry or -1 if not found.

X509_NAME_entry_count() returns the total number of entries.

X509_NAME_get_entry() returns an *X509_NAME* pointer to the requested entry or *NULL* if the index is invalid.

SEE ALSO

ERR_get_error (3), d2i_X509_*NAME* (3)

HISTORY

None.

X509_NAME_print_ex

NAME

X509_NAME_print_ex, X509_NAME_print_ex_fp, X509_NAME_print, X509_NAME_oneline – X509_NAME printing routines.

Synopsis

```
#include <openssl/x509.h>
int X509_NAME_print_ex(BIO *out, X509_NAME *nm, int indent, unsigned long flags);
int X509_NAME_print_ex_fp(FILE *fp, X509_NAME *nm, int indent, unsigned long flags);
char *X509_NAME_oneline(X509_NAME *a,char *buf,int size);
int X509_NAME_print(BIO *bp, X509_NAME *name, int obase);
```

DESCRIPTION

X509_NAME_print_ex() prints a human readable version of *nm* to BIO *out*. Each line (for multiline formats) is indented by *indent* spaces. The output format can be extensively customised by use of the *flags* parameter.

X509_NAME_print_ex_fp() is identical to X509_NAME_print_ex() except the output is written to FILE pointer *fp*.

X509_NAME_oneline() prints an ASCII version of *a* to *buf*. At most *size* bytes will be written. If *buf* is *NULL* then a buffer is dynamically allocated and returned, otherwise *buf* is returned.

X509_NAME_print() prints out *name* to *bp* indenting each line by *obase* characters. Multiple lines are used if the output (including indent) exceeds 80 characters.

NOTES

The functions X509_NAME_oneline() and X509_NAME_print() are legacy functions which produce a non standard output form, they don't handle multi character fields and have various quirks and inconsistencies. Their use is strongly discouraged in new applications.

Although there are a large number of possible flags for most purposes *XN_FLAG_ONELINE*, *XN_FLAG_MULTILINE* or *XN_FLAG_RFC2253* will suffice. As noted on the ASN1_*STRING_print_ex* (3) manual page for UTF8 terminals the *ASN1_STRFLAGS_ESC_MSB* should be unset: so for example *XN_FLAG_ONELINE & ~ASN1_STRFLAGS_ESC_MSB* would be used.

The complete set of the flags supported by X509_NAME_print_ex() is listed below.

Several options can be ored together.

The options *XN_FLAG_SEP_COMMA_PLUS*, *XN_FLAG_SEP_CPLUS_SPC*, *XN_FLAG_SEP_SPLUS_SPC* and *XN_FLAG_SEP_MULTILINE* determine the field separators to use. Two distinct separators are used between distinct RelativeDistinguishedName components and separate values in the same RDN for a multi-valued RDN. Multi-valued RDNs are currently very rare so the second separator will hardly ever be used.

XN_FLAG_SEP_COMMA_PLUS uses comma and plus as separators. *XN_FLAG_SEP_CPLUS_SPC* uses comma and plus with spaces: this is more readable that plain comma and plus. *XN_FLAG_SEP_SPLUS_SPC* uses spaced semicolon and plus. *XN_FLAG_SEP_MULTILINE* uses spaced newline and plus respectively.

If *XN_FLAG_DN_REV* is set the whole DN is printed in reversed order.

The fields *XN_FLAG_FN_SN*, *XN_FLAG_FN_LN*, *XN_FLAG_FN_OID*, *XN_FLAG_FN_NONE* determine how a field name is displayed. It will use the short name (e.g. CN) the long name (e.g. commonName) always use OID numerical form (normally OIDs are only used if the field name is not recognised) and no field name respectively.

If *XN_FLAG_SPC_EQ* is set then spaces will be placed around the '=' character separating field names and values.

If *XN_FLAG_DUMP_UNKNOWN_FIELDS* is set then the encoding of unknown fields is printed instead of the values.

If *XN_FLAG_FN_ALIGN* is set then field names are padded to 20 characters: this is only of use for multiline format.

Additionally all the options supported by ASN1_STRING_print_ex() can be used to control how each field value is displayed.

In addition a number options can be set for commonly used formats.

XN_FLAG_RFC2253 sets options which produce an output compatible with RFC2253 it is equivalent to: *ASN1_STRFLGS_RFC2253* | *XN_FLAG_SEP_COMMA_PLUS* | *XN_FLAG_DN_REV* | *XN_FLAG_FN_SN* | *XN_FLAG_DUMP_UNKNOWN_FIELDS*

XN_FLAG_ONELINE is a more readable one line format it is the same as: *ASN1_STRFLGS_RFC2253* | *ASN1_STRFLGS_ESC_QUOTE* | *XN_FLAG_SEP_CPLUS_SPC* | *XN_FLAG_SPC_EQ* | *XN_FLAG_FN_SN*

XN_FLAG_MULTILINE is a multiline format is is the same as: *ASN1_STRFLGS_ESC_CTRL* | *ASN1_STRFLGS_ESC_MSB* | *XN_FLAG_SEP_MULTILINE* | *XN_FLAG_SPC_EQ* | *XN_FLAG_FN_LN* | *XN_FLAG_FN_ALIGN*

XN_FLAG_COMPAT uses a format identical to X509_NAME_print(): in fact it calls X509_NAME_print() internally.

SEE ALSO

ASN1_*STRING_print_ex* (3)

HISTORY

None.

X509_new

NAME

X509_new, X509_free – X509 certificate ASN1 allocation functions

Synopsis

```
X509 *X509_new(void);
void X509_free(X509 *a);
```

DESCRIPTION

The X509 ASN1 allocation routines, allocate and free an X509 structure, which represents an X509 certificate.

X509_new() allocates and initializes a X509 structure.

X509_free() frees up the *X509* structure *a*.

RETURN VALUES

If the allocation fails, X509_new() returns *NULL* and sets an error code that can be obtained by *ERR_get_error* (3). Otherwise it returns a pointer to the newly allocated structure.

X509_free() returns no value.

SEE ALSO

ERR_get_error (3), `d2i_X509(3)`

HISTORY

X509_new() and X509_free() are available in all versions of SSLeay and OpenSSL.

SSL Application Programming Interface (API) Reference

This reference section includes the OpenSSL **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`

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.

---

d2i_SSL_SESSION

NAME

d2i_SSL_SESSION, i2d_SSL_SESSION – convert SSL_SESSION object from/to 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

d2i_SSL_SESSION() 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.

i2d_SSL_SESSION() 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()ed parts, it can therefore not 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. The reference count is 1, so that the session must be explicitly removed using *SSL_SESSION_free* (3), unless the SSL_SESSION object is completely taken over, when being called inside the get_session_cb() (see *SSL_CTX_sess_set_get_cb* (3)).

SSL_SESSION objects keep internal link information about the session cache list, when being inserted into one SSL_CTX object's session cache. One SSL_SESSION object, regardless of its reference count, must therefore only be used with one SSL_CTX object (and the SSL objects created from this SSL_CTX object).

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

d2i_SSL_SESSION() 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.

i2d_SSL_SESSION() 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

ssl (3), *SSL_SESSION_free* (3), *SSL_CTX_sess_set_get_cb* (3)

SSL

NAME

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* (3).

Then an SSL_CTX object is created as a framework to establish TLS/SSL enabled connections (see *SSL_CTX_new* (3)). 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* (3), *SSL_set_fd* (3) or *SSL_set_bio* (3) can be used to associate the network connection with the object.

Then the TLS/SSL handshake is performed using *SSL_accept* (3) or *SSL_connect* (3) respectively. *SSL_read* (3) and *SSL_write* (3) are used to read and write data on the TLS/SSL connection. *SSL_shutdown* (3) can be used to shut down the TLS/SSL connection.

DATA STRUCTURES

Currently the OpenSSL ssl library functions deals with the following data structures:

- SSL_METHOD (SSL Method)

  That's a dispatch structure describing the internal ssl library methods/functions which implement the various protocol versions (SSLv1, SSLv2 and TLSv1). It's needed to create an SSL_CTX.

- SSL_CIPHER (SSL Cipher)

  This structure 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)

  That's the global context structure which is created by a server or client once per program life-time and which holds mainly default values for the SSL structures which are later created for the connections.

- SSL_SESSION (SSL Session)

  This is a structure containing the current TLS/SSL session details for a connection: SSL_CIPHERs, client and server certificates, keys, etc.

- SSL (SSL Connection)

  That's the main SSL/TLS structure which is created by a server or client per established connection. This actually is the core structure in the SSL API. Under run-time the application usually deals with this structure which has links to mostly all other structures.

HEADER FILES

Currently the OpenSSL ssl library provides the following C header files containing the prototypes for the data structures and and functions:

- ssl.h

  That's the common header file for the SSL/TLS API. Include it into your program to make the API of the ssl library available. It internally includes both more private SSL headers and headers from the crypto library. Whenever you need hard-core details on the internals of the SSL API, look inside this header file.

- ssl2.h

  That's the sub header file dealing with the SSLv2 protocol only. *Usually you don't have to include it explicitly because it's already included by ssl.h.*

- ssl3.h

  That's the sub header file dealing with the SSLv3 protocol only. *Usually you don't have to include it explicitly because it's already included by ssl.h.*

- ssl23.h

  That's the sub header file dealing with the combined use of the SSLv2 and SSLv3 protocols. *Usually you don't have to include it explicitly because it's already included by ssl.h.*

- tls1.h

  That's the sub header file dealing with the TLSv1 protocol only. *Usually you don't have to include it explicitly because it's already included by ssl.h.*

API FUNCTIONS

Currently the OpenSSL ssl library exports 214 API functions. They are documented in the following:

DEALING WITH PROTOCOL METHODS

Here we document the various API functions which deal with the SSL/TLS protocol methods defined in SSL_METHOD structures.

- 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

Here we document the various API functions which deal with the SSL/TLS ciphers defined in SSL_CIPHER structures.

- 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* as a string. These are the various strings defined by the *SSL2_TXT_xxx*, *SSL3_TXT_xxx* and *TLS1_TXT_xxx* definitions in the header files.

- char *SSL_CIPHER_get_version(SSL_CIPHER *cipher);

  Returns a string like "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

Here we document the various API functions which deal with the SSL/TLS protocol context defined in the SSL_CTX structure.

- 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)(), 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_msg_callback(SSL_CTX *ctx, void (*cb)(int write_p, int version, int content_type, const void *buf, size_t len, SSL *ssl, void *arg));
- void SSL_CTX_set_msg_callback_arg (SSL_CTX *ctx, void *arg);
- 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 B<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 C<export> flag will be set if the reason for needing a temp key is that an export ciphersuite is in use, in which case, C<keylength> will contain the required keylength in bits. Generate a key of appropriate size (using ???) 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

Here we document the various API functions which deal with the SSL/TLS sessions defined in the SSL_SESSION structures.

- 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

Here we document the various API functions which deal with the SSL/TLS connection defined in the SSL structure.

- 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_msg_callback(SSL *ctx, void (*cb)(int write_p, int version, int content_type, const void *buf, size_t len, SSL *ssl, void *arg));
- void SSL_set_msg_callback_arg(SSL *ctx, void *arg);

- 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);

SEE ALSO

openssl (1), *crypto* (3), *SSL_accept* (3), *SSL_clear* (3), *SSL_connect* (3), *SSL_CIPHER_get_name* (3), *SSL_COMP_add_compression_method* (3), *SSL_CTX_add_extra_chain_cert* (3), *SSL_CTX_add_session* (3), *SSL_CTX_ctrl* (3), *SSL_CTX_flush_sessions* (3), *SSL_CTX_get_ex_new_index* (3), *SSL_CTX_get_verify_mode* (3), *SSL_CTX_load_verify_locations* (3) *SSL_CTX_new* (3), *SSL_CTX_sess_number* (3), *SSL_CTX_sess_set_cache_size* (3), *SSL_CTX_sess_set_get_cb* (3), *SSL_CTX_sessions* (3), *SSL_CTX_set_cert_store* (3), *SSL_CTX_set_cert_verify_callback* (3), *SSL_CTX_set_cipher_list* (3), *SSL_CTX_set_client_CA_list* (3), *SSL_CTX_set_client_cert_cb* (3), *SSL_CTX_set_default_passwd_cb* (3), *SSL_CTX_set_generate_session_id* (3), *SSL_CTX_set_info_callback* (3), *SSL_CTX_set_max_cert_list* (3), *SSL_CTX_set_mode* (3), *SSL_CTX_set_msg_callback* (3), *SSL_CTX_set_options* (3), *SSL_CTX_set_quiet_shutdown* (3), *SSL_CTX_set_session_cache_mode* (3), *SSL_CTX_set_session_id_context* (3), *SSL_CTX_set_ssl_version* (3), *SSL_CTX_set_timeout* (3), *SSL_CTX_set_tmp_rsa_callback* (3), *SSL_CTX_set_tmp_dh_callback* (3), *SSL_CTX_set_verify* (3), *SSL_CTX_use_certificate* (3), *SSL_alert_type_string* (3), *SSL_do_handshake* (3), *SSL_get_SSL_CTX* (3), *SSL_get_ciphers* (3), *SSL_get_client_CA_list* (3), *SSL_get_default_timeout* (3), *SSL_get_error* (3), *SSL_get_ex_data_X509* (3), *_STORE_CTX_idx* (3), *SSL_get_ex_new_index* (3), *SSL_get_fd* (3), *SSL_get_peer_cert_chain* (3), *SSL_get_rbio* (3), *SSL_get_session* (3), *SSL_get_verify_result* (3), *SSL_get_version* (3), *SSL_library_init* (3), *SSL_load_client_CA_file* (3), *SSL_new* (3), *SSL_pending* (3), *SSL_read* (3), *SSL_rstate_string* (3), *SSL_session_reused* (3), *SSL_set_bio* (3), *SSL_set_connect_state* (3), *SSL_set_fd* (3), *SSL_set_session* (3), *SSL_set_shutdown* (3), *SSL_shutdown* (3), *SSL_state_string* (3), *SSL_want* (3), *SSL_write* (3), *SSL_SESSION_free* (3), *SSL_SESSION_get_ex_new_index* (3), *SSL_SESSION_get_time* (3), *i_SSL_SESSION* (3)

HISTORY

The *ssl* (3) document appeared in OpenSSL 0.9.2

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

SSL_accept() 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 behaviour of SSL_accept() depends on the underlying BIO.

If the underlying BIO is blocking, SSL_accept() will only return once the handshake has been finished or an error occurred, except for SGC (Server Gated Cryptography). For SGC, SSL_accept() may return with -1, but SSL_get_error() will yield SSL_ERROR_WANT_READ/WRITE and SSL_accept() should be called again.

If the underlying BIO is non-blocking, SSL_accept() will also return when the underlying BIO could not satisfy the needs of SSL_accept() to continue the handshake, indicating the problem by the return value -1. In this case a call to SSL_get_error() 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 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 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 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 of action is need to continue the operation for non-blocking BIOs. Call SSL_get_error() with the return value ret to find out the reason.

SEE ALSO

SSL_get_error (3), *SSL_connect* (3), *SSL_shutdown* (3), *ssl* (3), *bio* (3), *SSL_set_connect_state* (3), *SSL_do_handshake* (3), *SSL_CTX_new* (3)

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>
const char *SSL_alert_type_string(int value);
const char *SSL_alert_type_string_long(int value);
const char *SSL_alert_desc_string(int value);
const char *SSL_alert_desc_string_long(int value);
```

DESCRIPTION

SSL_alert_type_string() returns a one letter string indicating the type of the alert specified by value.

SSL_alert_type_string_long() returns a string indicating the type of the alert specified by value.

SSL_alert_desc_string() returns a two letter string as a short form describing the reason of the alert specified by value .

SSL_alert_desc_string_long() 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 results 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 for non-fatal errors are certificate errors ("certificate expired", "unsupported certificate"), for which a warning alert may be sent. (The sending party may however decide to send a fatal error.) The receiving side may cancel the connection on reception of a warning alert on it discretion.

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 following strings can occur for SSL_alert_type_string() or SSL_alert_type_string_long():

- "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 SSL_alert_desc_string() or SSL_alert_desc_string_long():

- "CN"/"close notify"

  The connection shall 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 has failed 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 canceled"

  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

ssl (3), *SSL_CTX_set_info_callback* (3)

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

SSL_CIPHER_get_name() returns a pointer to the name of cipher. If the argument is the NULL pointer, a pointer to the constant value "NONE" is returned.

SSL_CIPHER_get_bits() 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.

SSL_CIPHER_get_version() returns the protocol version for cipher, currently "SSLv2", "SSLv3", or "TLSv1". If cipher is NULL, "(NONE)" is returned.

SSL_CIPHER_description() returns a textual description of the cipher used into the buffer buf of length len provided. 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 OPENSSL_malloc(). 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 like e.g. EXP-RC4-MD5 has only 40 secret bits. The algorithm does use the full 128 bits (which would be returned for alg_bits), of which however 88bits are fixed. The search space is hence only 40 bits.

The string returned by SSL_CIPHER_description() 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, DHDSS, 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.

EXAMPLES

Some examples for the output of SSL_CIPHER_description():

```
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
```

Restrictions

If SSL_CIPHER_description() is called with cipher being NULL, the library crashes.

If SSL_CIPHER_description() cannot handle a built-in cipher, the according description of the cipher property is unknown. This case should not occur.

RETURN VALUES

See DESCRIPTION

SEE ALSO

ssl (3), *SSL_get_current_cipher* (3), *SSL_get_ciphers* (3), *ciphers* (1)

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

SSL_clear 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 *SSL_shutdown* (3) was not called for the connection or at least *SSL_set_shutdown* (3) was used to set the SSL_SENT_SHUTDOWN state.

If a session was closed cleanly, the session object will be kept and all settings corresponding. This explicitly means, that e.g. the special method used during the session will be kept for the next handshake. So if the session was a TLSv1 session, a SSL client object will use a TLSv1 client method for the next handshake and a SSL server object will use a TLSv1 server method, even if SSLv23_*_methods were chosen on startup. This will might lead to connection failures (see *SSL_new* (3)) for a description of the method's properties.

WARNINGS

SSL_clear() resets the SSL object to allow for another connection. The reset operation however keeps several settings of the last sessions (some of these settings were made automatically during the last handshake). It only makes sense when opening a new session (or reusing an old one) with the same peer that shares these settings. SSL_clear() is not a short form for the sequence *SSL_free* (3); *SSL_new* (3); .

RETURN VALUES

The following return values can occur:

- 0

  The SSL_clear() operation could not be performed. Check the error stack to find out the reason.

- 1

  The SSL_clear() operation was successful.

SEE ALSO

SSL_new (3), *SSL_free* (3), *SSL_shutdown* (3), *SSL_set_shutdown* (3), *SSL_CTX_set_options* (3), *ssl* (3), *SSL_CTX_set_client_cert_cb* (3)

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

SSL_COMP_add_compression_method() 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. The TLS RFC does however not specify compression methods or their corresponding identifiers, so there is currently no compatible way to integrate compression with unknown peers. It is therefore currently not recommended to integrate 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.

WARNINGS

Once the identities of the compression methods for the TLS protocol have been standardized, the compression API will most likely be changed. Using it in the current state is not recommended.

RETURN VALUES

SSL_COMP_add_compression_method() may return the following values:

- 1

  The operation succeeded.

- 0

  The operation failed. Check the error queue to find out the reason.

SEE ALSO

ssl (3)

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

SSL_connect() 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 behaviour of SSL_connect() depends on the underlying BIO.

If the underlying BIO is blocking, SSL_connect() will only return once the handshake has been finished or an error occurred.

If the underlying BIO is non-blocking, SSL_connect() will also return when the underlying BIO could not satisfy the needs of SSL_connect() to continue the handshake, indicating the problem by the return value -1. 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 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 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 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 of action is need to continue the operation for non-blocking BIOs. Call SSL_get_error() with the return value ret to find out the reason.

SEE ALSO

SSL_get_error (3), *SSL_accept* (3), *SSL_shutdown* (3), *ssl* (3), *bio* (3), *SSL_set_connect_state* (3), *SSL_do_handshake* (3), *SSL_CTX_new* (3)

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

SSL_CTX_add_extra_chain_cert() 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* (3).

RETURN VALUES

SSL_CTX_add_extra_chain_cert() returns 1 on success. Check out the error stack to find out the reason for failure otherwise.

SEE ALSO

ssl (3), *SSL_CTX_use_certificate* (3), *SSL_CTX_set_client_cert_cb* (3), *SSL_CTX_load_verify_locations* (3)

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

SSL_CTX_add_session() 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* (3).

SSL_CTX_remove_session() removes the session c from the context ctx. *SSL_SESSION_free* (3) is called once for c.

SSL_add_session() and SSL_remove_session() are synonyms for their SSL_CTX_*() 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 actually identical (the SSL_SESSION object is identical), SSL_CTX_add_session() is a no-op, and the return value is 0.

If a server SSL_CTX is configured with the SSL_SESS_CACHE_NO_INTERNAL_STORE flag then the internal cache will not be populated automatically by new sessions negotiated by the SSL/TLS implementation, even though the internal cache will be searched automatically for session-resume requests (the latter can be surpressed by SSL_SESS_CACHE_NO_INTERNAL_LOOKUP). So the application can use SSL_CTX_add_session() directly to have full control over the sessions that can be resumed if desired.

RETURN VALUES

The following values are returned by all functions:

- 0

  The operation failed. In case of the add operation, it was tried to add the same (identical) session twice. In case of the remove operation, the session was not found in the cache.

- 1

  The operation succeeded.

SEE ALSO

ssl (3), *SSL_CTX_set_session_cache_mode* (3), *SSL_SESSION_free* (3)

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, void *parg);
long SSL_CTX_callback_ctrl(SSL_CTX *, int cmd, void (*fp)());
long SSL_ctrl(SSL *ssl, int cmd, long larg, void *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 command cmd 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*_ctrl() functions depend on the command supplied via the cmd parameter.

SEE ALSO

ssl (3)

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

SSL_CTX_flush_sessions() causes a run through the session cache of ctx to remove sessions expired at time tm.

SSL_flush_sessions() is a synonym for SSL_CTX_flush_sessions().

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 ones they are expired, they should be removed from the cache to save resources. This can either be done automatically whenever 255 new sessions were established (see *SSL_CTX_set_session_cache_mode* (3)) or manually by calling SSL_CTX_flush_sessions().

The parameter tm specifies the time which should be used for the expiration test, in most cases the actual time given by *time* (0) will be used.

SSL_CTX_flush_sessions() will only check sessions stored in the internal cache. When a session is found and removed, the remove_session_cb is however called to synchronize with the external cache (see *SSL_CTX_sess_set_get_cb* (3)).

RETURN VALUES

None.

SEE ALSO

ssl (3), *SSL_CTX_set_session_cache_mode* (3), *SSL_CTX_set_timeout* (3), *SSL_CTX_sess_set_get_cb* (3)

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

SSL_CTX_free() 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.

It also calls the free()ing procedures for indirectly affected items, if applicable: the session cache, the list of ciphers, the list of Client CAs, the certificates and keys.

WARNINGS

If a session-remove callback is set (SSL_CTX_sess_set_remove_cb()), this callback will be called for each session being freed from ctx's session cache. This implies, that all corresponding sessions from an external session cache are removed as well. If this is not desired, the user should explicitly unset the callback by calling SSL_CTX_sess_set_remove_cb(ctx, NULL) prior to calling SSL_CTX_free().

RETURN VALUES

SSL_CTX_free() does not provide diagnostic information.

SEE ALSO

SSL_CTX_new (3), *ssl* (3), *SSL_CTX_sess_set_get_cb* (3)

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.

SSL_CTX_get_ex_new_index() is used to register a new index for application specific data.

SSL_CTX_set_ex_data() is used to store application data at arg for idx into the ctx object.

SSL_CTX_get_ex_data() is used to retrieve the information for idx from ctx.

A detailed description for the *_get_ex_new_index() functionality can be found in *RSA_get_ex_new_index* (3). The *_get_ex_data() and *_set_ex_data() functionality is described in *CRYPTO_set_ex_data* (3).

SEE ALSO

ssl (3), *RSA_get_ex_new_index* (3), *CRYPTO_set_ex_data* (3)

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

SSL_CTX_get_verify_mode() returns the verification mode currently set in ctx.

SSL_get_verify_mode() returns the verification mode currently set in ssl.

SSL_CTX_get_verify_depth() 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.

SSL_get_verify_depth() 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.

SSL_CTX_get_verify_callback() 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.

SSL_get_verify_callback() 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

ssl (3), *SSL_CTX_set_verify* (3)

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

SSL_CTX_load_verify_locations() 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

```
-----BEGIN CERTIFICATE-----
... (CA certificate in base64 encoding) ...
-----END CERTIFICATE-----
```

sequences. Before, between, and after the certificates text is allowed which can be used e.g. for descriptions of 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 hence 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, e.g. 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 of which it will accept client certificates. This list is not influenced by the contents of CAfile or CApath and must explicitly be set using the *SSL_CTX_set_client_CA_list* (3) 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* (3), *SSL_CTX_use_certificate* (3).

WARNINGS

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. Make sure to not have expired certificates mixed with valid ones.

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 .
```

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.

SEE ALSO

ssl (3), *SSL_CTX_set_client_CA_list* (3), *SSL_get_client_CA_list* (3), *SSL_CTX_use_certificate* (3), *SSL_CTX_add_extra_chain_cert* (3), *SSL_CTX_set_cert_store* (3)

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

SSL_CTX_new() 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. 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.

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 e.g. SSLv23_server_method() and be able to negotiate with all possible clients, but to only allow newer protocols like SSLv3 or TLSv1.

SSL_CTX_new() 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 find out the reason.

- Pointer to an SSL_CTX object

  The return value points to an allocated SSL_CTX object.

SEE ALSO

SSL_CTX_free (3), *SSL_accept* (3), *ssl* (3), *SSL_set_connect_state* (3)

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_hits, SSL_CTX_sess_cb_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

SSL_CTX_sess_number() returns the current number of sessions in the internal session cache.

SSL_CTX_sess_connect() returns the number of started SSL/TLS handshakes in client mode.

SSL_CTX_sess_connect_good() returns the number of successfully established SSL/TLS sessions in client mode.

SSL_CTX_sess_connect_renegotiate() returns the number of start renegotiations in client mode.

SSL_CTX_sess_accept() returns the number of started SSL/TLS handshakes in server mode.

SSL_CTX_sess_accept_good() returns the number of successfully established SSL/TLS sessions in server mode.

SSL_CTX_sess_accept_renegotiate() returns the number of start renegotiations in server mode.

SSL_CTX_sess_hits() returns the number of successfully reused sessions. In client mode a session set with *SSL_set_session* (3) successfully reused is counted as a hit. In server mode a session successfully retrieved from internal or external cache is counted as a hit.

SSL_CTX_sess_cb_hits() returns the number of successfully retrieved sessions from the external session cache in server mode.

SSL_CTX_sess_misses() returns the number of sessions proposed by clients that were not found in the internal session cache in server mode.

SSL_CTX_sess_timeouts() 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.

SSL_CTX_sess_cache_full() returns the number of sessions that were removed because the maximum session cache size was exceeded.

RETURN VALUES

The functions return the values indicated in the DESCRIPTION section.

SEE ALSO

ssl (3), *SSL_set_session* (3), *SSL_CTX_set_session_cache_mode* (3) *SSL_CTX_sess_set_cache_size* (3)

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

SSL_CTX_sess_set_cache_size() sets the size of the internal session cache of context ctx to t.

SSL_CTX_sess_get_cache_size() returns the currently valid session cache size.

NOTES

The internal session cache size is SSL_SESSION_CACHE_MAX_SIZE_DEFAULT, currently 1024*20, so that up to 20000 sessions can be held. This size can be modified using the SSL_CTX_sess_set_cache_size() call. A special case is the size 0, which is used for unlimited size.

When the maximum number of sessions is reached, no more new sessions are added to the cache. New space may be added by calling *SSL_CTX_flush_sessions* (3) to remove expired sessions.

If the size of the session cache is reduced and more sessions are already in the session cache, old session will be removed at the next time a session shall be added. This removal is not synchronized with the expiration of sessions.

RETURN VALUES

SSL_CTX_sess_set_cache_size() returns the previously valid size.

SSL_CTX_sess_get_cache_size() returns the currently valid size.

SEE ALSO

ssl (3), *SSL_CTX_set_session_cache_mode* (3), *SSL_CTX_sess_number* (3), *SSL_CTX_flush_sessions* (3)

SSL_CTX_sess_set_new_cb

NAME

SSL_CTX_sess_set_new_cb, SSL_CTX_sess_set_remove_cb, SSL_CTX_sess_set_get_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

SSL_CTX_sess_set_new_cb() sets the callback function, which is automatically called whenever a new session was negotiated.

SSL_CTX_sess_set_remove_cb() 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.

SSL_CTX_sess_set_get_cb() sets the callback function which is called, whenever a SSL/TLS client proposed to resume a session but the session could not be found in the internal session cache (see *SSL_CTX_set_session_cache_mode* (3)). (SSL/TLS server only.)

SSL_CTX_sess_get_new_cb(), SSL_CTX_sess_get_remove_cb(), and SSL_CTX_sess_get_get_cb() allow to 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 d2 *i_SSL_SESSION* (3) interface.

The new_session_cb() is called, whenever a new session has been negotiated and session caching is enabled (see *SSL_CTX_set_session_cache_mode* (3)). The new_session_cb() is passed the ssl connection and the ssl session sess. If the callback returns 0, the session will be immediately removed again.

The remove_session_cb() is called, whenever the SSL engine removes a session from the internal cache. This happens when the session is removed because it is expired or when a connection was not shutdown cleanly. It also happens for all sessions in the internal session cache when *SSL_CTX_free* (3) is called. The remove_session_cb() is passed the ctx and the ssl session sess. It does not provide any feedback.

The get_session_cb() is only called on SSL/TLS servers with the session id proposed by the client. The get_session_cb() is always called, also when session caching was disabled. The get_session_cb() is passed the ssl onnection, 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, Normally the reference count is not incremented and therefore the session must not be explicitly freed with *SSL_SESSION_free* (3).

SEE ALSO

ssl (3), d2*i_SSL_SESSION* (3), *SSL_CTX_set_session_cache_mode* (3), *SSL_CTX_flush_sessions* (3), *SSL_SESSION_free* (3), *SSL_CTX_free* (3)

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

SSL_CTX_sessions() 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* (3) type database. It is possible to directly access this database e.g. for searching. In parallel, the sessions form a linked list which is maintained separately from the *lhash* (3) operations, so that the database must not be modified directly but by using the *SSL_CTX_add_session* (3) family of functions.

SEE ALSO

ssl (3), *lhash* (3), *SSL_CTX_add_session* (3), *SSL_CTX_set_session_cache_mode* (3)

SSL_CTX_set_cert_store

NAME

SSL_CTX_set_cert_store, SSL_CTX_get_cert_store – manipulate X509 certificate verification storage

Synopsis

```
#include <openssl/ssl.h>
void SSL_CTX_set_cert_store(SSL_CTX *ctx, X509_STORE *store);
X509_STORE *SSL_CTX_get_cert_store(SSL_CTX *ctx);
```

DESCRIPTION

SSL_CTX_set_cert_store() sets/replaces the certificate verification storage of ctx to/with store. If another X509_STORE object is currently set in ctx, it will be X509_STORE_free()ed.

SSL_CTX_get_cert_store() returns a pointer to the current certificate verification storage.

NOTES

In order to verify the certificates presented by the peer, trusted CA certificates must be accessed. These CA certificates are made available via lookup methods, handled inside the X509_STORE. From the X509_STORE the X509_STORE_CTX used when verifying certificates is created.

Typically the trusted certificate store is handled indirectly via using *SSL_CTX_load_verify_locations* (3). Using the SSL_CTX_set_cert_store() and SSL_CTX_get_cert_store() functions it is possible to manipulate the X509_STORE object beyond the *SSL_CTX_load_verify_locations* (3) call.

Currently no detailed documentation on how to use the X509_STORE object is available. Not all members of the X509_STORE are used when the verification takes place. So will e.g. the verify_callback() be overridden with the verify_callback() set via the *SSL_CTX_set_verify* (3) family of functions. This document must therefore be updated when documentation about the X509_STORE object and its handling becomes available.

RETURN VALUES

SSL_CTX_set_cert_store() does not return diagnostic output.

SSL_CTX_get_cert_store() returns the current setting.

SEE ALSO

ssl (3), *SSL_CTX_load_verify_locations* (3), *SSL_CTX_set_verify* (3)

SSL_CTX_set_cert_verify_callback

NAME

SSL_CTX_set_cert_verify_callback – set peer certificate verification procedure

Synopsis

```
#include <openssl/ssl.h>
void SSL_CTX_set_cert_verify_callback(SSL_CTX *ctx, int (*callback)(X509_STORE_CTX *,void
*), void *arg);
```

DESCRIPTION

SSL_CTX_set_cert_verify_callback() sets the verification callback function for *ctx*. SSL objects that are created from *ctx* inherit the setting valid at the time when *SSL_new* (3) is called.

NOTES

Whenever a certificate is verified during a SSL/TLS handshake, a verification function is called. If the application does not explicitly specify a verification callback function, the built-in verification function is used. If a verification callback *callback* is specified via SSL_CTX_set_cert_verify_callback(), the supplied callback function is called instead. By setting *callback* to NULL, the default behaviour is restored.

When the verification must be performed, *callback* will be called with the arguments callback(X509_STORE_CTX *x509_store_ctx, void *arg). The argument *arg* is specified by the application when setting *callback*.

callback should return 1 to indicate verification success and 0 to indicate verification failure. If SSL_VERIFY_PEER is set and *callback* returns 0, the handshake will fail. As the verification procedure may allow to continue the connection in case of failure (by always returning 1) the verification result must be set in any case using the error member of *x509_store_ctx* so that the calling application will be informed about the detailed result of the verification procedure!

Within *x509_store_ctx*, *callback* has access to the *verify_callback* function set using *SSL_CTX_set_verify* (3).

WARNINGS

Do not mix the verification callback described in this function with the verify_callback function called during the verification process. The latter is set using the *SSL_CTX_set_verify* (3) family of functions.

Providing a complete verification procedure including certificate purpose settings etc is a complex task. The built-in procedure is quite powerful and in most cases it should be sufficient to modify its behaviour using the verify_callback function.

RETURN VALUES

SSL_CTX_set_cert_verify_callback() does not provide diagnostic information.

SEE ALSO

ssl (3), *SSL_CTX_set_verify* (3), *SSL_get_verify_result* (3), *SSL_CTX_load_verify_locations* (3)

HISTORY

Previous to OpenSSL 0.9.7, the *arg* argument to SSL_CTX_set_cert_verify_callback was ignored, and *callback* was called simply as int (*callback)(X509_STORE_CTX *) To compile software written for previous versions of OpenSSL, a dummy argument will have to be added to *callback*.

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

SSL_CTX_set_cipher_list() sets the list of available ciphers for ctx using the control string str. The format of the string is described in *ciphers* (1). The list of ciphers is inherited by all ssl objects created from ctx.

SSL_set_cipher_list() 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 simply ignored. Failure is only flagged if no ciphers could be collected at all.

It should be noted, that 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 don't need a certificate, but DH-parameters must have been set. All other ciphers need a corresponding certificate and key.

A RSA cipher can only be chosen, when a RSA certificate is available. RSA export ciphers with a keylength of 512 bits for the RSA key require a temporary 512 bit RSA key, as typically the supplied key has a length of 1024 bit (see *SSL_CTX_set_tmp_rsa_callback* (3)). RSA ciphers using EDH need a certificate and key and additional DH-parameters (see *SSL_CTX_set_tmp_dh_callback* (3)).

A DSA cipher can only be chosen, when a DSA certificate is available. DSA ciphers always use DH key exchange and therefore need DH-parameters (see *SSL_CTX_set_tmp_dh_callback* (3)).

When these conditions are not met for any cipher in the list (e.g. a client only supports export RSA ciphers with a asymmetric key length of 512 bits and the server is not configured to use temporary RSA keys), the "no shared cipher" (SSL_R_NO_SHARED_CIPHER) error is generated and the handshake will fail.

RETURN VALUES

SSL_CTX_set_cipher_list() and SSL_set_cipher_list() return 1 if any cipher could be selected and 0 on complete failure.

SEE ALSO

ssl (3), *SSL_get_ciphers* (3), *SSL_CTX_use_certificate* (3), *SSL_CTX_set_tmp_rsa_callback* (3), *SSL_CTX_set_tmp_dh_callback* (3), *ciphers* (1)

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

SSL_CTX_set_client_CA_list() sets the list of CAs sent to the client when requesting a client certificate for ctx.

SSL_set_client_CA_list() 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.

SSL_CTX_add_client_CA() adds the CA name extracted from cacert to the list of CAs sent to the client when requesting a client certificate for ctx.

SSL_add_client_CA() 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 must explicitly be set using SSL_CTX_set_client_CA_list() for ctx and SSL_set_client_CA_list() 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 *SSL_CTX_load_verify_locations* (3) to additionally load them for verification.

If the list of acceptable CAs is compiled in a file, the *SSL_load_client_CA_file* (3) function can be used to help importing the necessary data.

SSL_CTX_add_client_CA() and SSL_add_client_CA() can be used to add additional items 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

SSL_CTX_set_client_CA_list() and SSL_set_client_CA_list() do not return diagnostic information.

SSL_CTX_add_client_CA() and SSL_add_client_CA() 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 out 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

ssl (3), *SSL_get_client_CA_list* (3), *SSL_load_client_CA_file* (3), *SSL_CTX_load_verify_locations* (3)

SSL_CTX_set_client_cert_cb

NAME

SSL_CTX_set_client_cert_cb, SSL_CTX_get_client_cert_cb – handle client certificate callback function

Synopsis

```
#include <openssl/ssl.h>
void SSL_CTX_set_client_cert_cb(SSL_CTX *ctx, int (*client_cert_cb)(SSL *ssl, X509 **x509,
EVP_PKEY **pkey));
int (*SSL_CTX_get_client_cert_cb(SSL_CTX *ctx))(SSL *ssl, X509 **x509, EVP_PKEY **pkey);
int (*client_cert_cb)(SSL *ssl, X509 **x509, EVP_PKEY **pkey);
```

DESCRIPTION

SSL_CTX_set_client_cert_cb() sets the client_cert_cb() callback, that is called when a client certificate is requested by a server and no certificate was yet set for the SSL object.

When client_cert_cb() is NULL, no callback function is used.

SSL_CTX_get_client_cert_cb() returns a pointer to the currently set callback function.

client_cert_cb() is the application defined callback. If it wants to set a certificate, a certificate/private key combination must be set using the x509 and pkey arguments and "1" must be returned. The certificate will be installed into ssl, see the NOTES and Restrictions sections. If no certificate should be set, "0" has to be returned and no certificate will be sent. A negative return value will suspend the handshake and the handshake function will return immediatly. *SSL_get_error* (3) will return SSL_ERROR_WANT_X509_LOOKUP to indicate, that the handshake was suspended. The next call to the handshake function will again lead to the call of client_cert_cb(). It is the job of the client_cert_cb() to store information about the state of the last call, if required to continue.

NOTES

During a handshake (or renegotiation) a server may request a certificate from the client. A client certificate must only be sent, when the server did send the request.

When a certificate was set using the *SSL_CTX_use_certificate* (3) family of functions, it will be sent to the server. The TLS standard requires that only a certificate is sent, if it matches the list of acceptable CAs sent by the server. This constraint is violated by the default behavior of the OpenSSL library. Using the callback function it is possible to implement a proper selection routine or to allow a user interaction to choose the certificate to be sent.

If a callback function is defined and no certificate was yet defined for the SSL object, the callback function will be called. If the callback function returns a certificate, the OpenSSL library will try to load the private key and certificate data into the SSL object using the SSL_use_certificate() and SSL_use_private_key() functions. Thus it will permanently install the certificate and key for this SSL object. It will not be reset by calling *SSL_clear* (3). If the callback returns no certificate, the OpenSSL library will not send a certificate.

Restrictions

The client_cert_cb() cannot return a complete certificate chain, it can only return one client certificate. If the chain only has a length of 2, the root CA certificate may be omitted according to the TLS standard and thus a standard conforming answer can be sent to the server. For a longer chain, the client must send the complete

chain (with the option to leave out the root CA certificate). This can only be accomplished by either adding the intermediate CA certificates into the trusted certificate store for the SSL_CTX object (resulting in having to add CA certificates that otherwise maybe would not be trusted), or by adding the chain certificates using the *SSL_CTX_add_extra_chain_cert* (3) function, which is only available for the SSL_CTX object as a whole and that therefore probably can only apply for one client certificate, making the concept of the callback function (to allow the choice from several certificates) questionable.

Once the SSL object has been used in conjunction with the callback function, the certificate will be set for the SSL object and will not be cleared even when *SSL_clear* (3) is being called. It is therefore mandatory to destroy the SSL object using *SSL_free* (3) and create a new one to return to the previous state.

SEE ALSO

ssl (3), *SSL_CTX_use_certificate* (3), *SSL_CTX_add_extra_chain_cert* (3), *SSL_get_client_CA_list* (3), *SSL_clear* (3), *SSL_free* (3)

SSL_CTX_set_default_passwd_cb

NAME

SSL_CTX_set_default_passwd_cb, SSL_CTX_set_default_passwd_cb_userdata – set passwd 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

SSL_CTX_set_default_passwd_cb() sets the default password callback called when loading/storing a PEM certificate with encryption.

SSL_CTX_set_default_passwd_cb_userdata() sets a pointer to userdata which will be provided to the password callback on invocation.

The pem_passwd_cb(), 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. 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 may 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.

When asking for the password interactively, pem_passwd_cb() can use rwflag to check, whether an item shall be encrypted (rwflag=1). In this case the password dialog may ask for the same password twice for comparison in order to catch typos, that would make decryption impossible.

Other items in PEM formatting (certificates) can also be encrypted, it is however not usual, as certificate information is considered public.

RETURN VALUES

SSL_CTX_set_default_passwd_cb() and SSL_CTX_set_default_passwd_cb_userdata() 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

ssl (3), *SSL_CTX_use_certificate* (3)

SSL_CTX_set_generate_session_id

NAME

SSL_CTX_set_generate_session_id, SSL_set_generate_session_id, SSL_has_matching_session_id – manipulate generation of SSL session IDs (server only)

Synopsis

```
#include <openssl/ssl.h>
typedef int (*GEN_SESSION_CB)(const SSL *ssl, unsigned char *id, unsigned int *id_len);
int SSL_CTX_set_generate_session_id(SSL_CTX *ctx, GEN_SESSION_CB cb);
int SSL_set_generate_session_id(SSL *ssl, GEN_SESSION_CB, cb);
int SSL_has_matching_session_id(const SSL *ssl, const unsigned char *id, unsigned int
id_len);
```

DESCRIPTION

SSL_CTX_set_generate_session_id() sets the callback function for generating new session ids for SSL/TLS sessions for ctx to be cb.

SSL_set_generate_session_id() sets the callback function for generating new session ids for SSL/TLS sessions for ssl to be cb.

SSL_has_matching_session_id() checks, whether a session with id id (of length id_len) is already contained in the internal session cache of the parent context of ssl.

NOTES

When a new session is established between client and server, the server generates a session id. The session id is an arbitrary sequence of bytes. The length of the session id is 16 bytes for SSLv2 sessions and between 1 and 32 bytes for SSLv3/TLSv1. The session id is not security critical but must be unique for the server. Additionally, the session id is transmitted in the clear when reusing the session so it must not contain sensitive information.

Without a callback being set, an OpenSSL server will generate a unique session id from pseudo random numbers of the maximum possible length. Using the callback function, the session id can be changed to contain additional information like e.g. a host id in order to improve load balancing or external caching techniques.

The callback function receives a pointer to the memory location to put id and a pointer to the maximum allowed length id_len. The buffer at location id is only guaranteed to have the size id_len. The callback is only allowed to generate a shorter id and reduce id_len; the callback must never increase id_len or write to the location id exceeding the given limit.

If a SSLv2 session id is generated and id_len is reduced, it will be restored after the callback has finished and the session id will be padded with 0x00. It is not recommended to change the id_len for SSLv2 sessions. The callback can use the *SSL_get_version* (3) function to check, whether the session is of type SSLv2.

The location id is filled with 0x00 before the callback is called, so the callback may only fill part of the possible length and leave id_len untouched while maintaining reproducibility.

Since the sessions must be distinguished, session ids must be unique. Without the callback a random number is used, so that the probability of generating the same session id is extremely small (2^128 possible ids for an SSLv2 session, 2^256 for SSLv3/TLSv1). In order to assure the uniqueness of the generated session id, the callback must call SSL_has_matching_session_id() and generate another id if a conflict occurs. If an id conflict

is not resolved, the handshake will fail. If the application codes e.g. a unique host id, a unique process number, and a unique sequence number into the session id, uniqueness could easily be achieved without randomness added (it should however be taken care that no confidential information is leaked this way). If the application can not guarantee uniqueness, it is recommended to use the maximum id_len and fill in the bytes not used to code special information with random data to avoid collisions.

SSL_has_matching_session_id() will only query the internal session cache, not the external one. Since the session id is generated before the handshake is completed, it is not immediately added to the cache. If another thread is using the same internal session cache, a race condition can occur in that another thread generates the same session id. Collisions can also occur when using an external session cache, since the external cache is not tested with SSL_has_matching_session_id() and the same race condition applies.

When calling SSL_has_matching_session_id() for an SSLv2 session with reduced id_len, the match operation will be performed using the fixed length required and with a 0x00 padded id.

The callback must return 0 if it cannot generate a session id for whatever reason and return 1 on success.

EXAMPLES

The callback function listed will generate a session id with the server id given, and will fill the rest with pseudo random bytes:

```
const char session_id_prefix = "www-18";

#define MAX_SESSION_ID_ATTEMPTS 10
static int generate_session_id(const SSL *ssl, unsigned char *id,
                               unsigned int *id_len)
     {
     unsigned int count = 0;
     const char *version;

     version = SSL_get_version(ssl);
     if (!strcmp(version, "SSLv2"))
 /* we must not change id_len */;

     do      {
             RAND_pseudo_bytes(id, *id_len);
             /* Prefix the session_id with the required prefix. NB: If our
              * prefix is too long, clip it - but there will be worse effects
              * anyway, eg. the server could only possibly create 1 session
              * ID (ie. the prefix!) so all future session negotiations will
              * fail due to conflicts. */
             memcpy(id, session_id_prefix,
                     (strlen(session_id_prefix) < *id_len) ?
                     strlen(session_id_prefix) : *id_len);
             }
     while(SSL_has_matching_session_id(ssl, id, *id_len) &&
             (++count < MAX_SESSION_ID_ATTEMPTS));
     if(count >= MAX_SESSION_ID_ATTEMPTS)
             return 0;
     return 1;
     }
```

RETURN VALUES

SSL_CTX_set_generate_session_id() and SSL_set_generate_session_id() always return 1.

SSL_has_matching_session_id() returns 1 if another session with the same id is already in the cache.

SEE ALSO

ssl (3), *SSL_get_version* (3)

HISTORY

SSL_CTX_set_generate_session_id(), SSL_set_generate_session_id() and SSL_has_matching_session_id() have been introduced in OpenSSL 0.9.7.

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))();
```

DESCRIPTION

SSL_CTX_set_info_callback() 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, not callback function is used.

SSL_set_info_callback() 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.

SSL_CTX_get_info_callback() returns a pointer to the currently set information callback function for ctx.

SSL_get_info_callback() 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.

where 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 non-blocking 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 0x20

  Callback has been called because a handshake is finished.

The current state information can be obtained using the *SSL_state_string* (3) family of functions.

The ret information can be evaluated using the *SSL_alert_type_string* (3) family of functions.

RETURN VALUES

SSL_set_info_callback() does not provide diagnostic information.

SSL_get_info_callback() 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=where& ~SSL_ST_MASK;

if (w & SSL_ST_CONNECT) str="SSL_connect";
else if (w & SSL_ST_ACCEPT) str="SSL_accept";
else str="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=(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 == 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

ssl (3), *SSL_state_string* (3), *SSL_alert_type_string* (3)

SSL_CTX_set_max_cert_list

NAME

SSL_CTX_set_max_cert_list, SSL_CTX_get_max_cert_list, SSL_set_max_cert_list,

SSL_get_max_cert_list – manipulate allowed for the peer's certificate chain

Synopsis

```
#include <openssl/ssl.h>
long SSL_CTX_set_max_cert_list(SSL_CTX *ctx, long size);
long SSL_CTX_get_max_cert_list(SSL_CTX *ctx);
long SSL_set_max_cert_list(SSL *ssl, long size);
long SSL_get_max_cert_list(SSL *ctx);
```

DESCRIPTION

SSL_CTX_set_max_cert_list() sets the maximum size allowed for the peer's certificate chain for all SSL objects created from ctx to be <size> bytes. The SSL objects inherit the setting valid for ctx at the time *SSL_new* (3) is being called.

SSL_CTX_get_max_cert_list() returns the currently set maximum size for ctx.

SSL_set_max_cert_list() sets the maximum size allowed for the peer's certificate chain for ssl to be <size> bytes. This setting stays valid until a new value is set.

SSL_get_max_cert_list() returns the currently set maximum size for ssl.

NOTES

During the handshake process, the peer may send a certificate chain. The TLS/SSL standard does not give any maximum size of the certificate chain. The OpenSSL library handles incoming data by a dynamically allocated buffer. In order to prevent this buffer from growing without bounds due to data received from a faulty or malicious peer, a maximum size for the certificate chain is set.

The default value for the maximum certificate chain size is 100kB (30kB on the 16bit DOS platform). This should be sufficient for usual certificate chains (OpenSSL's default maximum chain length is 10, see *SSL_CTX_set_verify* (3), and certificates without special extensions have a typical size of 1-2kB).

For special applications it can be necessary to extend the maximum certificate chain size allowed to be sent by the peer, see e.g. the work on "Internet X.509 Public Key Infrastructure Proxy Certificate Profile" and "TLS Delegation Protocol" at http://www.ietf.org/ and http://www.globus.org/ .

Under normal conditions it should never be necessary to set a value smaller than the default, as the buffer is handled dynamically and only uses the memory actually required by the data sent by the peer.

If the maximum certificate chain size allowed is exceeded, the handshake will fail with a SSL_R_EXCESSIVE_MESSAGE_SIZE error.

RETURN VALUES

SSL_CTX_set_max_cert_list() and SSL_set_max_cert_list() return the previously set value.

SSL_CTX_get_max_cert_list() and SSL_get_max_cert_list() return the currently set value.

SEE ALSO

ssl (3), *SSL_new* (3), *SSL_CTX_set_verify* (3)

HISTORY

SSL*_set/get_max_cert_list() have been introduced in OpenSSL 0.9.7.

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

SSL_CTX_set_mode() adds the mode set via bitmask in mode to ctx. Options already set before are not cleared.

SSL_set_mode() adds the mode set via bitmask in mode to ssl. Options already set before are not cleared.

SSL_CTX_get_mode() returns the mode set for ctx.

SSL_get_mode() 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. Once SSL_write() returns with r, r bytes have been successfully written and the next call to SSL_write() must only send the n-r bytes left, imitating the behaviour of write().

- 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 take place during normal operation, a *SSL_read* (3) or *SSL_write* (3) 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 flag SSL_MODE_AUTO_RETRY will cause read/write operations to only return after the handshake and successful completion.

RETURN VALUES

SSL_CTX_set_mode() and SSL_set_mode() return the new mode bitmask after adding mode.

SSL_CTX_get_mode() and SSL_get_mode() return the current bitmask.

SEE ALSO

ssl (3), *SSL_read* (3), *SSL_write* (3)

HISTORY

SSL_MODE_AUTO_RETRY as been added in OpenSSL 0.9.6.

SSL_CTX_set_msg_callback

NAME

SSL_CTX_set_msg_callback, SSL_CTX_set_msg_callback_arg, SSL_set_msg_callback, SSL_get_msg_callback_arg – install callback for observing protocol messages

Synopsis

```
#include <openssl/ssl.h>
void SSL_CTX_set_msg_callback(SSL_CTX *ctx, void (*cb)(int write_p, int version, int
content_type, const void *buf, size_t len, SSL *ssl, void *arg));
void SSL_CTX_set_msg_callback_arg(SSL_CTX *ctx, void *arg);
void SSL_set_msg_callback(SSL_CTX *ctx, void (*cb)(int write_p, int version, int
content_type, const void *buf, size_t len, SSL *ssl, void *arg));
void SSL_set_msg_callback_arg(SSL_CTX *ctx, void *arg);
```

DESCRIPTION

SSL_CTX_set_msg_callback() or SSL_set_msg_callback() can be used to define a message callback function *cb* for observing all SSL/TLS protocol messages (such as handshake messages) that are received or sent. SSL_CTX_set_msg_callback_arg() and SSL_set_msg_callback_arg() can be used to set argument *arg* to the callback function, which is available for arbitrary application use.

SSL_CTX_set_msg_callback() and SSL_CTX_set_msg_callback_arg() specify default settings that will be copied to new SSL objects by *SSL_new* (3). SSL_set_msg_callback() and SSL_set_msg_callback_arg() modify the actual settings of an SSL object. Using a 0 pointer for *cb* disables the message callback.

When *cb* is called by the SSL/TLS library for a protocol message, the function arguments have the following meaning:

- *write_p*

  This flag is 0 when a protocol message has been received and 1 when a protocol message has been sent.

- *version*

  The protocol version according to which the protocol message is interpreted by the library. Currently, this is one of SSL2_VERSION , SSL3_VERSION and TLS1_VERSION (for SSL 2.0, SSL 3.0 and TLS 1.0, respectively).

- *content_type*

  In the case of SSL 2.0, this is always 0. In the case of SSL 3.0 or TLS 1.0, this is one of the ContentType values defined in the protocol specification (change_cipher_spec(20), alert(21), handshake(22); but never application_data(23) because the callback will only be called for protocol messages).

- *buf*, *len*

  buf points to a buffer containing the protocol message, which consists of *len* bytes. The buffer is no longer valid after the callback function has returned.

- *ssl*

  The SSL object that received or sent the message.

- *arg*

The user-defined argument optionally defined by SSL_CTX_set_msg_callback_arg() or SSL_set_msg_callback_arg().

NOTES

Protocol messages are passed to the callback function after decryption and fragment collection where applicable. (Thus record boundaries are not visible.)

If processing a received protocol message results in an error, the callback function may not be called. For example, the callback function will never see messages that are considered too large to be processed.

Due to automatic protocol version negotiation, *version* is not necessarily the protocol version used by the sender of the message: If a TLS 1.0 ClientHello message is received by an SSL 3.0-only server, *version* will be SSL3_VERSION.

SEE ALSO

ssl (3), *SSL_new* (3)

HISTORY

SSL_CTX_set_msg_callback(), SSL_CTX_set_msg_callback_arg(), SSL_set_msg_callback() and SSL_get_msg_callback_arg() were added in OpenSSL 0.9.7.

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

SSL_CTX_set_options() adds the options set via bitmask in options to ctx. Options already set before are not cleared!

SSL_set_options() adds the options set via bitmask in options to ssl. Options already set before are not cleared!

SSL_CTX_get_options() returns the options set for ctx.

SSL_get_options() returns the options set for ssl.

NOTES

The behaviour 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 but can never be reset.

SSL_CTX_set_options() and SSL_set_options() affect the (external) protocol behaviour of the SSL library. The (internal) behaviour of the API can be changed by using the similar *SSL_CTX_set_mode* (3) and SSL_set_mode() functions.

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. SSL_clear() 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 ok but it should be ok to use 32. According to the SSLv3 spec, one should use 32 bytes for the challenge when operating in SSLv2/v3 compatibility mode, but as mentioned above, this breaks this server so 16 bytes is the way to go.

- SSL_OP_NETSCAPE_REUSE_CIPHER_CHANGE_BUG

  ssl3.netscape.com:443, first a connection is established with RC4-MD5. If it is then resumed, we end up using 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 really shows up when connecting via SSLv2/v3 then reconnecting via SSLv3. The cipher list changes....

NEW INFORMATION. Try connecting with a cipher list of just DES-CBC-SHA:RC4-MD5. For some weird reason, each new connection uses RC4-MD5, but a re-connect tries to use DES-CBC-SHA. So netscape, when doing a re-connect, always takes the first cipher in the cipher list.

- 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_DONT_INSERT_EMPTY_FRAGMENTS

  Disables a countermeasure against a SSL 3.0/TLS 1.0 protocol vulnerability affecting CBC ciphers, which cannot be handled by some broken SSL implementations. This option has no effect for connections using other ciphers.

- SSL_OP_AL

  All of the above bug workarounds.

It is usually safe to use B<ssl_op_all> to enable the bug workaround options if compatibility with somewhat broken implementations is desired.

The following B<modifying> options are available:

- 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 a 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_SINGLE_DH_USE

  Always create a new key when using temporary/ephemeral DH parameters (see L<ssl_ctx_set_tmp_dh_callback>). This option must be used to prevent small subgroup attacks, when the DH parameters were not generated using "strong" primes (e.g. when using DSA-parameters, see *dhparam* (1)). If "strong" primes were used, it is not strictly necessary to generate a new DH key during each handshake but it is also recommended. SSL_OP_SINGLE_DH_USE should therefore be enabled whenever temporary/ephemeral DH parameters are used.

- SSL_OP_EPHEMERAL_RSA

  Always use ephemeral (temporary) RSA key when doing RSA operations (see *SSL_CTX_set_tmp_rsa_callback* (3)). According to the specifications this is only done, when a RSA key can only be used for signature operations (namely under export ciphers with restricted RSA keylength). By setting this option, ephemeral RSA keys are always used. This option breaks compatibility with the SSL/TLS specifications and may lead to interoperability problems with clients and should therefore never be used. Ciphers with EDH (ephemeral Diffie-Hellman) key exchange should be used instead.

- SSL_OP_CIPHER_SERVER_PREFERENCE

  When choosing a cipher, use the server's preferences instead of the client preferences. When not set, the SSL server will always follow the clients preferences. When set, the SSLv3/TLSv1 server will choose following its own preferences. Because of the different protocol, for SSLv2 the server will send his list of preferences to the client and the client chooses.

- SSL_OP_PKCS1_CHECK_1

  ...

- SSL_OP_PKCS1_CHECK_2

  ...

- SSL_OP_NETSCAPE_CA_DN_BUG

  If we 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_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.

- SSL_OP_NO_SESSION_RESUMPTION_ON_RENEGOTIATION

  When performing renegotiation as a server, always start a new session (i.e., session resumption requests are only accepted in the initial handshake). This option is not needed for clients.

RETURN VALUES

SSL_CTX_set_options() and SSL_set_options() return the new options bitmask after adding options.

SSL_CTX_get_options() and SSL_get_options() return the current bitmask.

SEE ALSO

ssl (3), *SSL_new* (3), *SSL_clear* (3), *SSL_CTX_set_tmp_dh_callback* (3), *SSL_CTX_set_tmp_rsa_callback* (3), *dhparam* (1)

HISTORY

SSL_OP_CIPHER_SERVER_PREFERENCE and
SSL_OP_NO_SESSION_RESUMPTION_ON_RENEGOTIATION have been added in OpenSSL 0.9.7.

SSL_OP_TLS_ROLLBACK_BUG has been added in OpenSSL 0.9.6 and was automatically enabled with SSL_OP_ALL. As of 0.9.7, it is no longer included in SSL_OP_ALL and must be explicitly set.

SSL_OP_DONT_INSERT_EMPTY_FRAGMENTS has been added in OpenSSL 0.9.6e. Versions up to OpenSSL 0.9.6c do not include the countermeasure that can be disabled with this option (in OpenSSL 0.9.6d, it was always enabled).

SSL_CTX_set_quiet_shutdown

NAME

SSL_CTX_set_quiet_shutdown, SSL_CTX_get_quiet_shutdown, SSL_set_quiet_shutdown, SSL_get_quiet_shutdown – manipulate shutdown behaviour

Synopsis

```
#include <openssl/ssl.h>
void SSL_CTX_set_quiet_shutdown(SSL_CTX *ctx, int mode);
int SSL_CTX_get_quiet_shutdown(SSL_CTX *ctx);
void SSL_set_quiet_shutdown(SSL *ssl, int mode);
int SSL_get_quiet_shutdown(SSL *ssl);
```

DESCRIPTION

SSL_CTX_set_quiet_shutdown() sets the "quiet shutdown" flag for ctx to be mode. SSL objects created from

SSL_CTX_get_quiet_shutdown() returns the "quiet shutdown" setting of ctx.

SSL_set_quiet_shutdown() sets the "quiet shutdown" flag for ssl to be mode. The setting stays valid until ssl is removed with *SSL_free* (3) or SSL_set_quiet_shutdown() is called again. It is not changed when *SSL_clear* (3) is called. mode may be 0 or 1.

SSL_get_quiet_shutdown() returns the "quiet shutdown" setting of ssl.

NOTES

Normally when a SSL connection is finished, the parties must send out "close notify" alert messages using *SSL_shutdown* (3) for a clean shutdown.

When setting the "quiet shutdown" flag to 1, *SSL_shutdown* (3) will set the internal flags to SSL_SENT_SHUTDOWN | SSL_RECEIVED_SHUTDOWN. (*SSL_shutdown* (3) then behaves like *SSL_set_shutdown* (3) called with SSL_SENT_SHUTDOWN | SSL_RECEIVED_SHUTDOWN.) The session is thus considered to be shutdown, but no "close notify" alert is sent to the peer. This behaviour violates the TLS standard.

The default is normal shutdown behaviour as described by the TLS standard.

RETURN VALUES

SSL_CTX_set_quiet_shutdown() and SSL_set_quiet_shutdown() do not return diagnostic information.

SSL_CTX_get_quiet_shutdown() and SSL_get_quiet_shutdown return the current setting.

SEE ALSO

ssl (3), *SSL_shutdown* (3), *SSL_set_shutdown* (3), *SSL_new* (3), *SSL_clear* (3), *SSL_free* (3)

SSL_CTX_set_session_cache_mode

NAME

SSL_CTX_set_session_cache_mode, SSL_CTX_get_session_cache_mode – enable/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

SSL_CTX_set_session_cache_mode() enables/disables session caching by setting the operational mode for ctx to <mode>.

SSL_CTX_get_session_cache_mode() 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 exactly one id. The server then either agrees to reuse the session or it starts a full handshake (to create a new session).

A server will lookup up the session in its internal session storage. If the session is not found in internal storage or lookups for the internal storage have been deactivated (SSL_SESS_CACHE_NO_INTERNAL_LOOKUP), 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* (3)).

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* (3) 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 to be reused, the server looks for the corresponding session in (first) the internal session cache (unless SSL_SESS_CACHE_NO_INTERNAL_LOOKUP is set), then (second) in the external cache if available. 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* (3) function. Since this may lead to a delay which cannot be controlled, the automatic flushing may be disabled and *SSL_CTX_flush_sessions* (3) can be called explicitly by the application.

- SSL_SESS_CACHE_NO_INTERNAL_LOOKUP

  By setting this flag, session-resume operations in an SSL/TLS server will not automatically look up sessions in the internal cache, even if sessions are automatically stored there. If external session caching callbacks are in use, this flag guarantees that all lookups are directed to the external cache. As automatic lookup only applies for SSL/TLS servers, the flag has no effect on clients.

- SSL_SESS_CACHE_NO_INTERNAL_STORE

  Depending on the presence of SSL_SESS_CACHE_CLIENT and/or SSL_SESS_CACHE_SERVER, sessions negotiated in an SSL/TLS handshake may be cached for possible reuse. Normally a new session is added to the internal cache as well as any external session caching (callback) that is configured for the SSL_CTX. This flag will prevent sessions being stored in the internal cache (though the application can add them manually using *SSL_CTX_add_session* (3)). Note: in any SSL/TLS servers where external caching is configured, any successful session lookups in the external cache (ie. for session-resume requests) would normally be copied into the local cache before processing continues - this flag prevents these additions to the internal cache as well.

- SSL_SESS_CACHE_NO_INTERNA

  Enable both SSL_SESS_CACHE_NO_INTERNAL_LOOKUP and SSL_SESS_CACHE_NO_INTERNAL_STORE at the same time.

The default mode is SSL_SESS_CACHE_SERVER.

RETURN VALUES

SSL_CTX_set_session_cache_mode() returns the previously set cache mode.

SSL_CTX_get_session_cache_mode() returns the currently set cache mode.

SEE ALSO

L<ssl>, *SSL_set_session* (3), *SSL_session_reused* (3), *SSL_CTX_add_session* (3), *SSL_CTX_sess_number* (3), *SSL_CTX_sess_set_cache_size* (3), *SSL_CTX_sess_set_get_cb* (3), *SSL_CTX_set_session_id_context* (3), *SSL_CTX_set_timeout* (3), *SSL_CTX_flush_sessions* (3)

HISTORY

SSL_SESS_CACHE_NO_INTERNAL_STORE and SSL_SESS_CACHE_NO_INTERNAL were introduced in OpenSSL 0.9.6h.

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

SSL_CTX_set_session_id_context() sets the context sid_ctx of length sid_ctx_len within which a session can be reused for the ctx object.

SSL_set_session_id_context() 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/importing sessions with i2d_SSL_SESSION/d2i_SSL_SESSION it would be possible, to re-import 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, it is therefore possible to use e.g. the name of the application and/or the hostname and/or 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.

WARNINGS

If the session id context is not set on an SSL/TLS server, stored sessions will not be reused but 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, as an OpenSSL server checks the session id context itself before reusing a session as described above.

RETURN VALUES

SSL_CTX_set_session_id_context() and SSL_set_session_id_context() 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

ssl (3)

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

SSL_CTX_set_ssl_version() sets a new default TLS/SSL method for SSL objects newly created from this ctx. SSL objects already created with *SSL_new* (3) are not affected, except when *SSL_clear* (3) is being called.

SSL_set_ssl_method() sets a new TLS/SSL method for a particular ssl object. It may be reset, when SSL_clear() is called.

SSL_get_ssl_method() returns a function pointer to the TLS/SSL method set in ssl.

NOTES

The available method choices are described in *SSL_CTX_new* (3).

When *SSL_clear* (3) 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 SSL_CTX_set_ssl_version() and SSL_set_ssl_method():

- 0

  The new choice failed, check the error stack to find out the reason.

- 1

  The operation succeeded.

SEE ALSO

SSL_CTX_new (3), *SSL_new* (3), *SSL_clear* (3), *ssl* (3), *SSL_set_connect_state* (3)

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

SSL_CTX_set_timeout() sets the timeout for newly created sessions for ctx to t. The timeout value t must be given in seconds.

SSL_CTX_get_timeout() 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* (3) family of functions.

Expired sessions are removed from the internal session cache, whenever *SSL_CTX_flush_sessions* (3) is called, either directly by the application or automatically (see *SSL_CTX_set_session_cache_mode* (3))

The default value for session timeout is decided on a per protocol basis, see *SSL_get_default_timeout* (3). All currently supported protocols have the same default timeout value of 300 seconds.

RETURN VALUES

SSL_CTX_set_timeout() returns the previously set timeout value.

SSL_CTX_get_timeout() returns the currently set timeout value.

SEE ALSO

ssl (3), *SSL_CTX_set_session_cache_mode* (3), *SSL_SESSION_get_time* (3), *SSL_CTX_flush_sessions* (3), *SSL_get_default_timeout* (3)

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

SSL_CTX_set_tmp_dh_callback() sets the callback function for ctx to be used when a DH parameters are required to tmp_dh_callback. The callback is inherited by all ssl objects created from ctx.

SSL_CTX_set_tmp_dh() sets DH parameters to be used to be dh. The key is inherited by all ssl objects created from ctx.

SSL_set_tmp_dh_callback() sets the callback only for ssl.

SSL_set_tmp_dh() 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* (3) 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 strictly 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 (see especially the section about DSA parameters below), 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, but it is not very large, so application authors/users should consider to always enable this option.

As 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* (1) application. In order to reduce the computer time needed for this generation, it is possible to use DSA parameters instead (see *dhparam* (1)), but in this case SSL_OP_SINGLE_DH_USE is mandatory.

Application authors may compile in DH parameters. Files dh512.pem, dh1024.pem, dh2048.pem, and dh4096 in the 'apps' directory of 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* (1) application. Authors may also generate their own set of parameters using *dhparam* (1), but a user may not be sure how the parameters were generated. The generation of DH parameters during installation is therefore 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.

EXAMPLES

Handle DH parameters for key lengths of 512 and 1024 bits. (Error handling partly left out.)

```
...
/* Set up ephemeral DH stuff */
DH *dh_512 = NULL;
DH *dh_1024 = NULL;
FILE *paramfile;

...
/* "openssl dhparam -out dh_param_512.pem -2 512" */
paramfile = fopen("dh_param_512.pem", "r");
if (paramfile) {
  dh_512 = PEM_read_DHparams(paramfile, NULL, NULL, NULL);
  fclose(paramfile);
}
/* "openssl dhparam -out dh_param_1024.pem -2 1024" */
paramfile = fopen("dh_param_1024.pem", "r");
if (paramfile) {
  dh_1024 = PEM_read_DHparams(paramfile, NULL, NULL, NULL);
  fclose(paramfile);
}
...

/* "openssl dhparam -C -2 512" etc... */
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:
      /* Generating a key on the fly is very costly, so use what is there */
      setup_dh_parameters_like_above();
    }
    return(dh_tmp);
}
```

RETURN VALUES

SSL_CTX_set_tmp_dh_callback() and SSL_set_tmp_dh_callback() do not return diagnostic output.

SSL_CTX_set_tmp_dh() and SSL_set_tmp_dh() do return 1 on success and 0 on failure. Check the error queue to find out the reason of failure.

SEE ALSO

ssl (3), *SSL_CTX_set_cipher_list* (3), *SSL_CTX_set_tmp_rsa_callback* (3), *SSL_CTX_set_options* (3), *ciphers* (1), *dhparam* (1)

SSL_CTX_set_tmp_rsa_callback

NAME

SSL_CTX_set_tmp_rsa_callback, SSL_CTX_set_tmp_rsa, SSL_CTX_need_tmp_rsa,

SSL_set_tmp_rsa_callback, SSL_set_tmp_rsa, SSL_need_tmp_rsa – handle RSA keys for ephemeral key exchange

Synopsis

```
#include <openssl/ssl.h>
void SSL_CTX_set_tmp_rsa_callback(SSL_CTX *ctx, RSA *(*tmp_rsa_callback)(SSL *ssl, int
is_export, int keylength));
long SSL_CTX_set_tmp_rsa(SSL_CTX *ctx, RSA *rsa);
long SSL_CTX_need_tmp_rsa(SSL_CTX *ctx);
void SSL_set_tmp_rsa_callback(SSL_CTX *ctx, RSA *(*tmp_rsa_callback)(SSL *ssl, int
is_export, int keylength));
long SSL_set_tmp_rsa(SSL *ssl, RSA *rsa) long SSL_need_tmp_rsa(SSL *ssl) RSA
*(*tmp_rsa_callback)(SSL *ssl, int is_export, int keylength));
```

DESCRIPTION

SSL_CTX_set_tmp_rsa_callback() sets the callback function for ctx to be used when a temporary/ephemeral RSA key is required to tmp_rsa_callback. The callback is inherited by all SSL objects newly created from ctx with <*SSL_new* (3)|*SSL_new* (3)>. Already created SSL objects are not affected.

SSL_CTX_set_tmp_rsa() sets the temporary/ephemeral RSA key to be used to be rsa. The key is inherited by all SSL objects newly created from ctx with <*SSL_new* (3)|*SSL_new* (3)>. Already created SSL objects are not affected.

SSL_CTX_need_tmp_rsa() returns 1, if a temporary/ephemeral RSA key is needed for RSA-based strength-limited 'exportable' ciphersuites because a RSA key with a keysize larger than 512 bits is installed.

SSL_set_tmp_rsa_callback() sets the callback only for ssl.

SSL_set_tmp_rsa() sets the key only for ssl .

SSL_need_tmp_rsa() returns 1, if a temporary/ephemeral RSA key is needed, for RSA-based strength-limited 'exportable' ciphersuites because a RSA key with a keysize larger than 512 bits is installed.

These functions apply to SSL/TLS servers only.

NOTES

When using a cipher with RSA authentication, an ephemeral RSA key exchange can take place. In this case the session data are negotiated using the ephemeral/temporary RSA key and the RSA key supplied and certified by the certificate chain is only used for signing.

Under previous export restrictions, ciphers with RSA keys shorter (512 bits) than the usual key length of 1024 bits were created. To use these ciphers with RSA keys of usual length, an ephemeral key exchange must be performed, as the normal (certified) key cannot be directly used.

Using ephemeral RSA key exchange yields forward secrecy, as the connection can only be decrypted, when the RSA key is known. By generating a temporary RSA 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) RSA key, as this key was used for signing only. The downside is that creating a RSA key is computationally expensive.

Additionally, the use of ephemeral RSA key exchange is only allowed in the TLS standard, when the RSA key can be used for signing only, that is for export ciphers. Using ephemeral RSA key exchange for other purposes violates the standard and can break interoperability with clients. It is therefore strongly recommended to not use ephemeral RSA key exchange and use EDH (Ephemeral Diffie-Hellman) key exchange instead in order to achieve forward secrecy (see *SSL_CTX_set_tmp_dh_callback* (3)).

On OpenSSL servers ephemeral RSA key exchange is therefore disabled by default and must be explicitly enabled using the SSL_OP_EPHEMERAL_RSA option of *SSL_CTX_set_options* (3), violating the TLS/SSL standard. When ephemeral RSA key exchange is required for export ciphers, it will automatically be used without this option!

An application may either directly specify the key or can supply the key via a callback function. The callback approach has the advantage, that the callback may generate the key only in case it is actually needed. As the generation of a RSA key is however costly, it will lead to a significant delay in the handshake procedure. Another advantage of the callback function is that it can supply keys of different size (e.g. for SSL_OP_EPHEMERAL_RSA usage) while the explicit setting of the key is only useful for key size of 512 bits to satisfy the export restricted ciphers and does give away key length if a longer key would be allowed.

The tmp_rsa_callback is called with the keylength needed and the is_export information. The is_export flag is set, when the ephemeral RSA key exchange is performed with an export cipher.

EXAMPLES

Generate temporary RSA keys to prepare ephemeral RSA key exchange. As the generation of a RSA key costs a lot of computer time, they saved for later reuse. For demonstration purposes, two keys for 512 bits and 1024 bits respectively are generated.

```
...
/* Set up ephemeral RSA stuff */
RSA *rsa_512 = NULL;
RSA *rsa_1024 = NULL;

rsa_512 = RSA_generate_key(512,RSA_F4,NULL,NULL);
if (rsa_512 == NULL)
    evaluate_error_queue();

rsa_1024 = RSA_generate_key(1024,RSA_F4,NULL,NULL);
if (rsa_1024 == NULL)
  evaluate_error_queue();

...

RSA *tmp_rsa_callback(SSL *s, int is_export, int keylength)
{
   RSA *rsa_tmp=NULL;

   switch (keylength) {
   case 512:
     if (rsa_512)
       rsa_tmp = rsa_512;
     else { /* generate on the fly, should not happen in this example */
```

```
        rsa_tmp = RSA_generate_key(keylength,RSA_F4,NULL,NULL);
        rsa_512 = rsa_tmp; /* Remember for later reuse */
      }
      break;
    case 1024:
      if (rsa_1024)
        rsa_tmp=rsa_1024;
      else
        should_not_happen_in_this_example();
      break;
    default:
      /* Generating a key on the fly is very costly, so use what is there */
      if (rsa_1024)
        rsa_tmp=rsa_1024;
      else
        rsa_tmp=rsa_512; /* Use at least a shorter key */
    }
    return(rsa_tmp);
}
```

RETURN VALUES

SSL_CTX_set_tmp_rsa_callback() and SSL_set_tmp_rsa_callback() do not return diagnostic output.

SSL_CTX_set_tmp_rsa() and SSL_set_tmp_rsa() do return 1 on success and 0 on failure. Check the error queue to find out the reason of failure.

SSL_CTX_need_tmp_rsa() and SSL_need_tmp_rsa() return 1 if a temporary RSA key is needed and 0 otherwise.

SEE ALSO

ssl (3), *SSL_CTX_set_cipher_list* (3), *SSL_CTX_set_options* (3), *SSL_CTX_set_tmp_dh_callback* (3), *SSL_new* (3), *ciphers* (1)

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_STORE_CTX
*));
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

SSL_CTX_set_verify() sets the verification flags for ctx to be mode and specifies the verify_callback function to be used. If no callback function shall be specified, the NULL pointer can be used for verify_callback.

SSL_set_verify() sets the verification flags for ssl to be mode and specifies the verify_callback function to be used. If no callback function shall be specified, the NULL pointer can be used for verify_callback. In this case last verify_callback set specifically for this ssl remains. If no special callback was set before, the default callback for the underlying ctx is used, that was valid at the the time ssl was created with *SSL_new* (3).

SSL_CTX_set_verify_depth() sets the maximum depth for the certificate chain verification that shall be allowed for ctx. (See the Restrictions section.)

SSL_set_verify_depth() sets the maximum depth for the certificate chain verification that shall be allowed for ssl. (See the Restrictions section.)

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* (3) 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, the TLS/SSL handshake is immediately terminated with an alert message containing the reason for the verification failure. The behaviour 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, 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

Exactly one of the mode flags SSL_VERIFY_NONE and SSL_VERIFY_PEER must be set at any time.

The actual verification procedure is performed either using the built-in verification procedure or using another application provided verification function set with *SSL_CTX_set_cert_verify_callback* (3). The following descriptions apply in the case of the built-in procedure. An application provided procedure also has access to the verify depth information and the verify_callback() function, but the way this information is used may be different.

SSL_CTX_set_verify_depth() and SSL_set_verify_depth() 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 behaviour 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). 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 and verify_callback is called with preverify_ok=0. By applying X509_CTX_store_* functions verify_callback can locate the certificate in question and perform additional steps (see EXAMPLES). 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 is immediately stopped with "verification failed" state. If SSL_VERIFY_PEER is set, a verification failure alert is sent to the peer and the TLS/SSL handshake is terminated. If verify_callback returns 1, the verification process is continued. If verify_callback always returns 1, the TLS/SSL handshake will not be terminated with respect to verification failures and the connection will be established. The calling process can however retrieve the error code of the last verification error using *SSL_get_verify_result* (3) 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 behaviour, if the SSL_VERIFY_PEER and SSL_VERIFY_NONE are not used as required (exactly one must be set at any time).

The certificate verification depth set with SSL[_CTX]_verify_depth() 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 realizes an example verify_callback function that will always continue the TLS/SSL handshake regardless of verification failure, if wished. The callback realizes a verification depth limit with more informational output.

All verification errors are printed, informations about the certificate chain are printed on request. The example is realized for a server that does allow but not require client certificates.

The example makes use of the ex_data technique to store application data into/retrieve application data from the SSL structure (see *SSL_get_ex_new_index* (3), *SSL_get_ex_data_X509* (3), *_STORE_CTX_idx* (3)).

```
...
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

ssl (3), *SSL_new* (3), *SSL_CTX_get_verify_mode* (3), *SSL_get_verify_result* (3), *SSL_CTX_load_verify_locations* (3), *SSL_get_peer_certificate* (3), *SSL_CTX_set_cert_verify_callback* (3), *SSL_get_ex_data_X509* (3), *_STORE_CTX_idx* (3), *SSL_get_ex_new_index* (3)

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 *SSL_new* (3) 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* (3) is called for this SSL object.

SSL_CTX_use_certificate() loads the certificate x into ctx, SSL_use_certificate() 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* (3) function.

SSL_CTX_use_certificate_ASN1() loads the ASN1 encoded certificate from the memory location d (with length len) into ctx, SSL_use_certificate_ASN1() loads the ASN1 encoded certificate into ssl.

SSL_CTX_use_certificate_file() 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. SSL_use_certificate_file() loads the certificate from file into ssl. See the NOTES section on why SSL_CTX_use_certificate_chain_file() should be preferred.

SSL_CTX_use_certificate_chain_file() loads a certificate chain from file into ctx . The certificates must be in PEM format and must be sorted starting with the subject's certificate (actual client or server certificate), followed by intermediate CA certificates if applicable, and ending at the highest level (root) CA. There is no corresponding function working on a single SSL object.

SSL_CTX_use_PrivateKey() adds pkey as private key to ctx. SSL_CTX_use_RSAPrivateKey() adds the private key rsa of type RSA to ctx. SSL_use_PrivateKey() adds pkey as private key to ssl; SSL_use_RSAPrivateKey() adds rsa as private key of type RSA to ssl.

SSL_CTX_use_PrivateKey_ASN1() adds the private key of type pk stored at memory location d (length len) to ctx. SSL_CTX_use_RSAPrivateKey_ASN1() adds the private key of type RSA stored at memory location d (length len) to ctx. SSL_use_PrivateKey_ASN1() and SSL_use_RSAPrivateKey_ASN1() add the private key to ssl.

SSL_CTX_use_PrivateKey_file() 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. SSL_CTX_use_RSAPrivateKey_file() adds the first private RSA key found in file to ctx. SSL_use_PrivateKey_file() adds the first private key found in file ssl; SSL_use_RSAPrivateKey_file() adds the first private RSA key found to ssl.

SSL_CTX_check_private_key() 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 e.g. the last item was a RSA certificate or key, the RSA key/certificate pair will be checked. SSL_check_private_key() performs the same check for ssl. If no key/certificate was explicitly added for this ssl, the last item added into ctx will be checked.

NOTES

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 also *SSL_CTX_set_cipher_list* (3).

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 SSL_CTX_use_certificate_chain_file() is only applicable to PEM formatting. Files of type SSL_FILETYPE_PEM can contain more than one item.

SSL_CTX_use_certificate_chain_file() adds the first certificate found in the file to the certificate store. The other certificates are added to the store of chain certificates using *SSL_CTX_add_extra_chain_cert* (3). 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 type of certificate at the same time, it is recommended to use the SSL_CTX_use_certificate_chain_file() instead of the SSL_CTX_use_certificate_file() function in order to allow 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* (3).

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* (3). (Certificate files might be encrypted as well from the technical point of view, it however does not make sense as the data in the certificate is considered public anyway.)

RETURN VALUES

On success, the functions return 1. Otherwise check out the error stack to find out the reason.

SEE ALSO

ssl (3), *SSL_new* (3), *SSL_clear* (3), *SSL_CTX_load_verify_locations* (3), *SSL_CTX_set_default_passwd_cb* (3), *SSL_CTX_set_cipher_list* (3), *SSL_CTX_set_client_cert_cb* (3), *SSL_CTX_add_extra_chain_cert* (3)

SSL_do_handshake

NAME

SSL_do_handshake – perform a TLS/SSL handshake

Synopsis

```
#include <openssl/ssl.h>
int SSL_do_handshake(SSL *ssl);
```

DESCRIPTION

SSL_do_handshake() will wait for a SSL/TLS handshake to take place. If the connection is in client mode, the handshake will be started. The handshake routines may have to be explicitly set in advance using either *SSL_set_connect_state* (3) or *SSL_set_accept_state* (3).

NOTES

The behaviour of SSL_do_handshake() depends on the underlying BIO.

If the underlying BIO is blocking, SSL_do_handshake() will only return once the handshake has been finished or an error occurred, except for SGC (Server Gated Cryptography). For SGC, SSL_do_handshake() may return with -1, but SSL_get_error() will yield SSL_ERROR_WANT_READ/WRITE and SSL_do_handshake() should be called again.

If the underlying BIO is non-blocking, SSL_do_handshake() will also return when the underlying BIO could not satisfy the needs of SSL_do_handshake() to continue the handshake. In this case a call to SSL_get_error() with the return value of SSL_do_handshake() 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_do_handshake(). 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 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 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 of action is need to continue the operation for non-blocking BIOs. Call SSL_get_error() with the return value ret to find out the reason.

SEE ALSO

SSL_get_error (3), *SSL_connect* (3), *SSL_accept* (3), *ssl* (3), *bio* (3), *SSL_set_connect_state* (3)

SSL_free

NAME

SSL_free – free an allocated SSL structure

Synopsis

```
#include <openssl/ssl.h>
void SSL_free(SSL *ssl);
```

DESCRIPTION

SSL_free() 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

SSL_free() also calls the free()ing 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 SSL_free(), as 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 SSL_free() and the internal session cache. If the session is considered bad, because *SSL_shutdown* (3) was not called for the connection and *SSL_set_shutdown* (3) 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

SSL_free() does not provide diagnostic information.

SSL_new (3), *SSL_clear* (3), *SSL_shutdown* (3), *SSL_set_shutdown* (3), *ssl* (3)

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);
```

DESCRIPTION

SSL_get_ciphers() returns the stack of available SSL_CIPHERs for ssl, sorted by preference. If ssl is NULL or no ciphers are available, NULL is returned.

SSL_get_cipher_list() 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.

NOTES

The details of the ciphers obtained by SSL_get_ciphers() can be obtained using the *SSL_CIPHER_get_name* (3) family of functions.

Call SSL_get_cipher_list() with priority starting from 0 to obtain the sorted list of available ciphers, until NULL is returned.

RETURN VALUES

See DESCRIPTION

SEE ALSO

ssl (3), *SSL_CTX_set_cipher_list* (3), *SSL_CIPHER_get_name* (3)

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

SSL_CTX_get_client_CA_list() returns the list of client CAs explicitly set for ctx using *SSL_CTX_set_client_CA_list* (3).

SSL_get_client_CA_list() returns the list of client CAs explicitly set for ssl using SSL_set_client_CA_list() or ssl's SSL_CTX object with *SSL_CTX_set_client_CA_list* (3), when in server mode. In client mode, SSL_get_client_CA_list returns the list of client CAs sent from the server, if any.

RETURN VALUES

SSL_CTX_set_client_CA_list() and SSL_set_client_CA_list() do not return diagnostic information.

SSL_CTX_add_client_CA() and SSL_add_client_CA() 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

ssl (3), *SSL_CTX_set_client_CA_list* (3), *SSL_CTX_set_client_cert_cb* (3)

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

SSL_get_current_cipher() returns a pointer to an SSL_CIPHER object containing the description of the actually used cipher of a connection established with the ssl object.

SSL_get_cipher() and SSL_get_cipher_name() are identical macros to obtain the name of the currently used cipher. SSL_get_cipher_bits() is a macro to obtain the number of secret/algorithm bits used and SSL_get_cipher_version() returns the protocol name. See *SSL_CIPHER_get_name* (3) for more details.

RETURN VALUES

SSL_get_current_cipher() returns the cipher actually used or NULL, when no session has been established.

SEE ALSO

ssl (3), *SSL_CIPHER_get_name* (3)

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

SSL_get_default_timeout() 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* (3), the hardcoded default timeout for the protocol will be used.

SSL_get_default_timeout() return this hardcoded value, which is 300 seconds for all currently supported protocols (SSLv2, SSLv3, and TLSv1).

RETURN VALUES

See Description.

SEE ALSO

ssl (3), *SSL_CTX_set_session_cache_mode* (3), *SSL_SESSION_get_time* (3), *SSL_CTX_flush_sessions* (3), *SSL_get_default_timeout* (3)

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

SSL_get_error() returns a result code (suitable for the C "switch" statement) for a preceding call to SSL_connect(), SSL_accept(), SSL_do_handshake(), SSL_read(), SSL_peek(), or SSL_write() on ssl. The value returned by that TLS/SSL I/O function must be passed to SSL_get_error() in parameter ret.

In addition to ssl and ret, SSL_get_error() inspects the current thread's OpenSSL error queue. Thus, SSL_get_error() 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 SSL_get_error() 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. Note that 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_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. Note that 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 SSL_set_fd() was used), select() or poll() on the underlying socket can be used to find out when the TLS/SSL I/O function should be retried.

  Caveat: Any TLS/SSL I/O function can lead to either of SSL_ERROR_WANT_READ and SSL_ERROR_WANT_WRITE . In particular, SSL_read() or SSL_peek() may want to write data and SSL_write() may want to read data. This is mainly because TLS/SSL handshakes may occur at any time during the protocol (initiated by either the client or the server); SSL_read(), SSL_peek(), and SSL_write() will handle any pending handshakes.

- SSL_ERROR_WANT_CONNECT, SSL_ERROR_WANT_ACCEPT

The operation did not complete; the same TLS/SSL I/O function should be called again later. The underlying BIO was not connected yet to the peer and the call would block in connect()/accept(). The SSL function should be called again when the connection is established. These messages can only appear with a BIO_s_connect() or BIO_s_accept() BIO, respectively. In order to find out, when the connection has been successfully established, on many platforms select() or poll() for writing on the socket file descriptor can be used.

- SSL_ERROR_WANT_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. The TLS/SSL I/O function should be called again later. Details depend on the application.

- SSL_ERROR_SYSCAL

  Some I/O error occurred. The OpenSSL error queue may contain more information on the error. If the error queue is empty (i.e. ERR_get_error() returns 0), B<ret> can be used to find out more about the error: If B<ret>, an EOF was observed that violates the protocol. If B<ret>, the underlying B<ret> reported an I/O error (for socket I/O on Unix systems, consult B<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.

SEE ALSO

err (3)

HISTORY

SSL_get_error() was added in SSLeay 0.8.

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

SSL_get_ex_data_X509_STORE_CTX_idx() returns the index number under which the pointer to the SSL object is stored into the X509_STORE_CTX object.

NOTES

Whenever a 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 SSL_get_ex_data_X509_STORE_CTX_idx(). The index value is set when SSL_get_ex_data_X509_STORE_CTX_idx() 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 SSL_get_ex_data_X509_STORE_CTX_idx() allows to access the SSL object for the connection to be accessed during the verify_callback() when checking the peers certificate. Please check the example in *SSL_CTX_set_verify* (3),

SEE ALSO

ssl (3), *SSL_CTX_set_verify* (3), *CRYPTO_set_ex_data* (3)

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.

SSL_get_ex_new_index() is used to register a new index for application specific data.

SSL_set_ex_data() is used to store application data at arg for idx into the ssl object.

SSL_get_ex_data() 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* (3). The *_get_ex_data() and *_set_ex_data() functionality is described in *CRYPTO_set_ex_data* (3).

EXAMPLES

An example on how to use the functionality is included in the example verify_callback() in *SSL_CTX_set_verify* (3).

SEE ALSO

ssl (3), *RSA_get_ex_new_index* (3), *CRYPTO_set_ex_data* (3), *SSL_CTX_set_verify* (3)

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

SSL_get_fd() returns the file descriptor which is linked to ssl. SSL_get_rfd() and SSL_get_wfd() return the file descriptors for the read or the write channel, which can be different. If the read and the write channel are different, SSL_get_fd() 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

SSL_set_fd (3), *ssl* (3) , *bio* (3)

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

SSL_get_peer_cert_chain() 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 *SSL_get_peer_certificate* (3). 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 any longer.

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

ssl (3), *SSL_get_peer_certificate* (3)

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

SSL_get_peer_certificate() returns a pointer to the X509 certificate the peer presented. If the peer did not present a certificate, NULL is returned.

NOTES

Due to the protocol definition, a TLS/SSL server will always send a certificate, if present. A client will only send a certificate when explicitly requested to do so by the server (see *SSL_CTX_set_verify* (3)). If an anonymous cipher is used, no certificates are sent.

That a certificate is returned does not indicate information about the verification state, use *SSL_get_verify_result* (3) 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 X509_free().

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

ssl (3), *SSL_get_verify_result* (3), *SSL_CTX_set_verify* (3)

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

SSL_get_rbio() and SSL_get_wbio() 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

SSL_set_bio (3), *ssl* (3) , *bio* (3)

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

SSL_get_session() 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.

SSL_get0_session() is the same as SSL_get_session().

SSL_get1_session() is the same as SSL_get_session(), but the reference count of the SSL_SESSION is incremented by one.

NOTES

The ssl session contains all information required to re-establish the connection without a new handshake.

SSL_get0_session() 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 *SSL_clear* (3) or *SSL_free* (3) is called, the session may 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* (3).

If the data is to be kept, SSL_get1_session() will increment the reference count, so that the session will not be implicitly removed by other operations but stays in memory. In order to remove the session *SSL_SESSION_free* (3) must be explicitly called once to decrement the reference count again.

SSL_SESSION objects keep internal link information about the session cache list, when being inserted into one SSL_CTX object's session cache. One SSL_SESSION object, regardless of its reference count, must therefore only be used with one SSL_CTX object (and the SSL objects created from this SSL_CTX object).

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

ssl (3), *SSL_free* (3), *SSL_clear* (3), *SSL_SESSION_free* (3)

SSL_get_SSL_CTX

NAME

SSL_get_SSL_CTX – get the SSL_CTX from which an SSL is created

Synopsis

```
#include <openssl/ssl.h>
SSL_CTX *SSL_get_SSL_CTX(SSL *ssl);
```

DESCRIPTION

SSL_get_SSL_CTX() returns a pointer to the SSL_CTX object, from which ssl was created with *SSL_new* (3).

RETURN VALUES

The pointer to the SSL_CTX object is returned.

SEE ALSO

ssl (3), *SSL_new* (3)

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

SSL_get_verify_result() returns the result of the verification of the X509 certificate presented by the peer, if any.

NOTES

SSL_get_verify_result() 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 however not indicate success. SSL_get_verify_result() is only useful in connection with *SSL_get_peer_certificate* (3).

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* (1).

SEE ALSO

ssl (3), *SSL_set_verify_result* (3), *SSL_get_peer_certificate* (3), *verify* (1)

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

SSL_get_cipher_version() 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

ssl (3)

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

SSL_library_init() registers the available ciphers and digests.

OpenSSL_add_ssl_algorithms() and SSLeay_add_ssl_algorithms() are synonyms for SSL_library_init().

NOTES

SSL_library_init() must be called before any other action takes place.

WARNING

SSL_library_init() only registers ciphers. Another important initialization is the seeding of the PRNG (Pseudo Random Number Generator), which has to be performed separately.

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();
```

RETURN VALUES

SSL_library_init() always returns "1", so it is safe to discard the return value.

SEE ALSO

ssl (3), *SSL_load_error_strings* (3), *RAND_add* (3)

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

SSL_load_client_CA_file() reads certificates from file and returns a STACK_OF(X509_NAME) with the subject names found.

NOTES

SSL_load_client_CA_file() 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 function for *SSL_CTX_set_client_CA_list* (3), 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

ssl (3), *SSL_CTX_set_client_CA_list* (3)

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

SSL_new() 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

SSL_free (3), *SSL_clear* (3), *SSL_CTX_set_options* (3), *SSL_get_SSL_CTX* (3), *ssl* (3)

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

SSL_pending() 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 *SSL_read* (3).

RETURN VALUES

The number of bytes pending is returned.

Restrictions

SSL_pending() takes into account only bytes from the TLS/SSL record that is currently 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 SSL_pending().

Up to OpenSSL 0.9.6, SSL_pending() does not check if the record type of pending data is application data.

SEE ALSO

SSL_read (3), *ssl* (3)

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

SSL_read() tries to read num bytes from the specified ssl into the buffer buf.

NOTES

If necessary, SSL_read() will negotiate a TLS/SSL session, if not already explicitly performed by *SSL_connect* (3) or *SSL_accept* (3). If the peer requests a re-negotiation, it will be performed transparently during the SSL_read() operation. The behaviour of SSL_read() depends on the underlying BIO.

For the transparent negotiation to succeed, the ssl must have been initialized to client or server mode. This is being done by calling *SSL_set_connect_state* (3) or SSL_set_accept_state() before the first call to an SSL_read() or *SSL_write* (3) function.

SSL_read() works 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, it can 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 retrieved 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, SSL_read() will 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 (e.g. 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, SSL_read() will only return, once the read operation has been finished or an error occurred, except when a renegotiation take place, in which case a SSL_ERROR_WANT_READ may occur. This behaviour can be controlled with the SSL_MODE_AUTO_RETRY flag of the *SSL_CTX_set_mode* (3) call.

If the underlying BIO is non-blocking, SSL_read() 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* (3) with the return value of SSL_read() will yield SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. As at any time a re-negotiation 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, like a BIO pair, data must be written into or retrieved out of the BIO before being able to continue.

WARNING

When an SSL_read() 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 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. The reason may either be a clean shutdown due to a "close notify" alert sent by the peer (in which case the SSL_RECEIVED_SHUTDOWN flag in the ssl shutdown state is set (see *SSL_shutdown* (3), *SSL_set_shutdown* (3)). It is also possible, that the peer simply shut down the underlying transport and the shutdown is incomplete. Call SSL_get_error() with the return value ret to find out, 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 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 out the reason.

SEE ALSO

SSL_get_error (3), *SSL_write* (3), *SSL_CTX_set_mode* (3), *SSL_CTX_new* (3), *SSL_connect* (3), *SSL_accept* (3) *SSL_set_connect_state* (3), *SSL_shutdown* (3), *SSL_set_shutdown* (3), *ssl* (3), *bio* (3)

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

SSL_rstate_string() returns a 2 letter string indicating the current read state of the SSL object ssl.

SSL_rstate_string_long() 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 should only seldom be needed in applications.

RETURN VALUES

SSL_rstate_string() and SSL_rstate_string_long() 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"/"unknown"

  The read state is unknown. This should never happen.

SEE ALSO

ssl (3)

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

SSL_SESSION_free() 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.

NOTES

SSL_SESSION objects are allocated, when a TLS/SSL handshake operation is successfully completed. Depending on the settings, see *SSL_CTX_set_session_cache_mode* (3), the SSL_SESSION objects are internally referenced by the SSL_CTX and linked into its session cache. SSL objects may be using the SSL_SESSION object; as a session may be reused, several SSL objects may be using one SSL_SESSION object at the same time. It is therefore crucial to keep the reference count (usage information) correct and not delete a SSL_SESSION object that is still used, as this may lead to program failures due to dangling pointers. These failures may also appear delayed, e.g. when an SSL_SESSION object was completely freed as the reference count incorrectly became 0, but it is still referenced in the internal session cache and the cache list is processed during a *SSL_CTX_flush_sessions* (3) operation.

SSL_SESSION_free() must only be called for SSL_SESSION objects, for which the reference count was explicitly incremented (e.g. by calling SSL_get1_session(), see *SSL_get_session* (3)) or when the SSL_SESSION object was generated outside a TLS handshake operation, e.g. by using d2*i_SSL_SESSION* (3). It must not be called on other SSL_SESSION objects, as this would cause incorrect reference counts and therefore program failures.

RETURN VALUES

SSL_SESSION_free() does not provide diagnostic information.

SEE ALSO

ssl (3), *SSL_get_session* (3), *SSL_CTX_set_session_cache_mode* (3), *SSL_CTX_flush_sessions* (3), d2*i_SSL_SESSION* (3)

SSL_SESSION_get_ex_new_index

NAME

SSL_SESSION_get_ex_new_index, SSL_SESSION_set_ex_data, SSL_SESSION_get_ex_data – internal 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(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.

SSL_SESSION_get_ex_new_index() is used to register a new index for application specific data.

SSL_SESSION_set_ex_data() is used to store application data at arg for idx into the session object.

SSL_SESSION_get_ex_data() 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* (3). The *_get_ex_data() and *_set_ex_data() functionality is described in *CRYPTO_set_ex_data* (3).

WARNINGS

The application data is only maintained for sessions held in memory. The application data is not included when dumping the session with i2d_SSL_SESSION() (and all functions indirectly calling the dump functions like PEM_write_SSL_SESSION() and PEM_write_bio_SSL_SESSION()) and can therefore not be restored.

SEE ALSO

ssl (3), *RSA_get_ex_new_index* (3), *CRYPTO_set_ex_data* (3)

SSL_SESSION_get_time

NAME

SSL_SESSION_get_time, SSL_SESSION_set_time, SSL_SESSION_get_timeout,

SSL_SESSION_get_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

SSL_SESSION_get_time() 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.

SSL_SESSION_set_time() replaces the creation time of the session s with the chosen value tm.

SSL_SESSION_get_timeout() returns the timeout value set for session s in seconds.

SSL_SESSION_set_timeout() 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 *SSL_CTX_set_timeout* (3). Using these functions it is possible to extend or shorten the lifetime of the session.

RETURN VALUES

SSL_SESSION_get_time() and SSL_SESSION_get_timeout() return the currently valid values.

SSL_SESSION_set_time() and SSL_SESSION_set_timeout() return 1 on success.

If any of the function is passed the NULL pointer for the session s, 0 is returned.

SEE ALSO

ssl (3), *SSL_CTX_set_timeout* (3), *SSL_get_default_timeout* (3)

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

Query, 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

ssl (3), *SSL_set_session* (3), *SSL_CTX_set_session_cache_mode* (3)

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

SSL_set_bio() 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 behaviour 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 , BIO_free() will be called (for both the reading and writing side, if different).

RETURN VALUES

SSL_set_bio() cannot fail.

SEE ALSO

SSL_get_rbio (3), *SSL_connect* (3), *SSL_accept* (3), *SSL_shutdown* (3), *ssl* (3), *bio* (3)

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

SSL_set_connect_state() sets ssl to work in client mode.

SSL_set_accept_state() sets ssl to work in server mode.

NOTES

When the SSL_CTX object was created with *SSL_CTX_new* (3), 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* (3) or SSL_set_ssl_method().)

When beginning a new handshake, the SSL engine must know whether it must call the connect (client) or accept (server) routines. Even though it may be clear from the method chosen, whether client or server mode was requested, the handshake routines must be explicitly set.

When using the *SSL_connect* (3) or *SSL_accept* (3) routines, the correct handshake routines are automatically set. When performing a transparent negotiation using *SSL_write* (3) or *SSL_read* (3), the handshake routines must be explicitly set in advance using either SSL_set_connect_state() or SSL_set_accept_state().

RETURN VALUES

SSL_set_connect_state() and SSL_set_accept_state() do not return diagnostic information.

SEE ALSO

ssl (3), *SSL_new* (3), *SSL_CTX_new* (3), *SSL_connect* (3), *SSL_accept* (3), *SSL_write* (3), *SSL_read* (3), *SSL_do_handshake* (3), *SSL_CTX_set_ssl_version* (3)

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

SSL_set_fd() sets the file descriptor fd as the input/output facility for the TLS/SSL (encrypted) side of ssl. 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 fd. The BIO and hence the SSL engine inherit the behaviour of fd. If fd is non-blocking, the ssl will also have non-blocking behaviour.

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

SSL_set_rfd() and SSL_set_wfd() 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

SSL_get_fd (3), *SSL_set_bio* (3), *SSL_connect* (3), *SSL_accept* (3), *SSL_shutdown* (3), *ssl* (3) , *bio* (3)

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

SSL_set_session() sets session to be used when the TLS/SSL connection is to be established. SSL_set_session() 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(). Whether the session was reused can be queried with the *SSL_session_reused* (3) call.

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.

NOTES

SSL_SESSION objects keep internal link information about the session cache list, when being inserted into one SSL_CTX object's session cache. One SSL_SESSION object, regardless of its reference count, must therefore only be used with one SSL_CTX object (and the SSL objects created from this SSL_CTX object).

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

ssl (3), *SSL_SESSION_free* (3), *SSL_get_session* (3), *SSL_session_reused* (3), *SSL_CTX_set_session_cache_mode* (3)

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

SSL_set_shutdown() sets the shutdown state of ssl to mode.

SSL_get_shutdown() 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* (3) or *SSL_free* (3) is called, it is considered bad and removed according to RFC2246. The actual condition for a correctly closed session is SSL_SENT_SHUTDOWN (according to the TLS RFC, it is acceptable to only send the "close notify" alert but to not wait for the peer's answer, when the underlying connection is closed). SSL_set_shutdown() can be used to set this state without sending a close alert to the peer (see *SSL_shutdown* (3)).

If a "close notify" was received, SSL_RECEIVED_SHUTDOWN will be set, for setting SSL_SENT_SHUTDOWN the application must however still call *SSL_shutdown* (3) or SSL_set_shutdown() itself.

RETURN VALUES

SSL_set_shutdown() does not return diagnostic information.

SSL_get_shutdown() returns the current setting.

SEE ALSO

ssl (3), *SSL_shutdown* (3), *SSL_CTX_set_quiet_shutdown* (3), *SSL_clear* (3), *SSL_free* (3)

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

SSL_set_verify_result() 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

SSL_set_verify_result() 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* (1).

RETURN VALUES

SSL_set_verify_result() does not provide a return value.

SEE ALSO

ssl (3), *SSL_get_verify_result* (3), *SSL_get_peer_certificate* (3), *verify* (1)

SSL_shutdown

NAME

SSL_shutdown – shut down a TLS/SSL connection

Synopsis

```
#include <openssl/ssl.h>
int SSL_shutdown(SSL *ssl);
```

DESCRIPTION

SSL_shutdown() shuts down an active TLS/SSL connection. It sends the "close notify" shutdown alert to the peer.

NOTES

SSL_shutdown() 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 shutdown procedure consists of 2 steps: the sending of the "close notify" shutdown alert and the reception of the peer's "close notify" shutdown alert. According to the TLS standard, it is acceptable for an application to only send its shutdown alert and then close the underlying connection without waiting for the peer's response (this way resources can be saved, as the process can already terminate or serve another connection). When the underlying connection shall be used for more communications, the complete shutdown procedure (bidirectional "close notify" alerts) must be performed, so that the peers stay synchronized.

SSL_shutdown() supports both uni- and bidirectional shutdown by its 2 step behaviour.

- When the application is the first party to send the "close notify"

  alert, SSL_shutdown() will only send the alert and the set the SSL_SENT_SHUTDOWN flag (so that the session is considered good and will be kept in cache). SSL_shutdown() will then return with 0. If a unidirectional shutdown is enough (the underlying connection shall be closed anyway), this first call to SSL_shutdown() is sufficient. In order to complete the bidirectional shutdown handshake, SSL_shutdown() must be called again. The second call will make SSL_shutdown() wait for the peer's "close notify" shutdown alert. On success, the second call to SSL_shutdown() will return with 1.

- If the peer already sent the "close notify" alert and it was already processed implicitly inside another function (*SSL_read* (3)), the SSL_RECEIVED_SHUTDOWN flag is set. SSL_shutdown() will send the "close notify" alert, set the SSL_SENT_SHUTDOWN flag and will immediately return with 1. Whether SSL_RECEIVED_SHUTDOWN is already set can be checked using the SSL_get_shutdown() (see also *SSL_set_shutdown* (3) call.

It is therefore recommended, to check the return value of SSL_shutdown() and call SSL_shutdown() again, if the bidirectional shutdown is not yet complete (return value of the first call is 0). As the shutdown is not specially handled in the SSLv2 protocol, SSL_shutdown() will succeed on the first call.

The behaviour of SSL_shutdown() additionally depends on the underlying BIO.

If the underlying BIO is blocking, SSL_shutdown() will only return once the handshake step has been finished or an error occurred.

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.

SSL_shutdown() can be modified to only set the connection to "shutdown" state but not actually send the "close notify" alert messages, see *SSL_CTX_set_quiet_shutdown* (3). When "quiet shutdown" is enabled, SSL_shutdown() will always succeed and return 1.

RETURN VALUES

The following return values can occur:

- 1

  The shutdown was successfully completed. The "close notify" alert was sent and the peer's "close notify" alert was received.

- 0

  The shutdown is not yet finished. Call SSL_shutdown() for a second time, if a bidirectional shutdown shall be performed. The output of *SSL_get_error* (3) may be misleading, as an erroneous SSL_ERROR_SYSCALL may be flagged even though no error occurred.

- -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 need to continue the operation for non-blocking BIOs. Call *SSL_get_error* (3) with the return value ret to find out the reason.

SEE ALSO

SSL_get_error (3), *SSL_connect* (3), *SSL_accept* (3), *SSL_set_shutdown* (3), *SSL_CTX_set_quiet_shutdown* (3), *SSL_clear* (3), *SSL_free* (3), *ssl* (3), *bio* (3)

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

SSL_state_string() returns a 6 letter string indicating the current state of the SSL object ssl.

SSL_state_string_long() 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. It however can be of significant interest during the handshake.

When using non-blocking sockets, the function call performing the handshake may return with SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE condition, so that SSL_state_string[_long]() may 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.

SEE ALSO

ssl (3), *SSL_CTX_set_info_callback* (3)

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

SSL_want() returns state information for the SSL object ssl.

The other SSL_want_*() calls are shortcuts for the possible states returned by SSL_want().

NOTES

SSL_want() examines the internal state information of the SSL object. Its return values are similar to that of *SSL_get_error* (3). Unlike *SSL_get_error* (3), which also evaluates the error queue, the results are obtained by examining an internal state flag only. The information must therefore only be used for normal operation under non-blocking I/O. Error conditions are not handled and must be treated using *SSL_get_error* (3).

The result returned by SSL_want() should always be consistent with the result of *SSL_get_error* (3).

RETURN VALUES

The following return values can currently 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* (3) 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* (3) 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* (3) should return SSL_ERROR_WANT_X509_LOOKUP.

SSL_want_nothing(), SSL_want_read(), SSL_want_write(), SSL_want_x509_lookup() return 1, when the corresponding condition is true or 0 otherwise.

SEE ALSO

ssl (3), *err* (3), *SSL_get_error* (3)

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

SSL_write() writes num bytes from the buffer buf into the specified ssl connection.

NOTES

If necessary, SSL_write() will negotiate a TLS/SSL session, if not already explicitly performed by *SSL_connect* (3) or *SSL_accept* (3). If the peer requests a re-negotiation, it will be performed transparently during the SSL_write() operation. The behaviour 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 being done by calling *SSL_set_connect_state* (3) or SSL_set_accept_state() before the first call to an *SSL_read* (3) or SSL_write() function.

If the underlying BIO is blocking, SSL_write() will only return, once the write operation has been finished or an error occurred, except when a renegotiation take place, in which case a SSL_ERROR_WANT_READ may occur. This behaviour can be controlled with the SSL_MODE_AUTO_RETRY flag of the *SSL_CTX_set_mode* (3) call.

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* (3) with the return value of SSL_write() will yield SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. As at any time a re-negotiation 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.

SSL_write() will only return with success, when the complete contents of buf of length num has been written. This default behaviour can be changed with the SSL_MODE_ENABLE_PARTIAL_WRITE option of *SSL_CTX_set_mode* (3). When this flag is set, SSL_write() will also return with success, when a partial write has been successfully completed. In this case the SSL_write() operation is considered completed. The bytes are sent and a new SSL_write() operation with a new buffer (with the already sent bytes removed) must be started. A partial write is performed with the size of a message block, which is 16kB for SSLv3/TLSv1.

WARNING

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.

When calling SSL_write() with num=0 bytes to be sent the behaviour is undefined.

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. Probably the underlying connection was closed. Call SSL_get_error() with the return value ret to find out, 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, why the closure happened.

- <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 out the reason.

SEE ALSO

SSL_get_error (3), *SSL_read* (3), *SSL_CTX_set_mode* (3), *SSL_CTX_new* (3), *SSL_connect* (3), *SSL_accept* (3) *SSL_set_connect_state* (3), *ssl* (3), *bio* (3)

A Data Structures and Header Files

This appendix lists the header files and the data structures included in HP SSL for OpenVMS.

A.1 Header Files

- SSL.H
- SSL2.H
- SSL23.H
- SSL3.H
- TLS.H
- SSL_EXAMPLES.H

A.2 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 */

};
```

A.3 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 */

};
```

A.4 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;
```

A.5 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;
```

A.6 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;
```

A.7 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;
};
```

A.8 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 New and Changed APIs in OpenSSL 0.9.7d and 0.9.7e

This appendix lists the new and changed APIs in OpenSSL 0.9.7d (included in HP SSL Version 1.2), and in OpenSSL 0.9.7e (inclued in HP SSL Version 1.3.).

B.1 New AES APIs in OpenSSL 0.9.7e

The following AES APIs are new in OpenSSL 0.9.7e and in HP SSL Version 1.3.

```
void AES_cfb1_encrypt(const unsigned char *in, unsigned char *out,
                          const unsigned long length, const AES_KEY *key,
                          unsigned char *ivec, int *num, const int enc);

void AES_cfb8_encrypt(const unsigned char *in, unsigned char *out,
                          const unsigned long length, const AES_KEY *key,
                          unsigned char *ivec, int *num, const int enc);

void AES_cfbr_encrypt_block(const unsigned char *in,unsigned char *out,
                          const int nbits,const AES_KEY *key,
                          unsigned char *ivec,const int enc);
```

B.2 New CRYPTO APIs in OpenSSL 0.9.7e

The following CRYPTO APIs are new in OpenSSL 0.9.7e and in HP SSL Version 1.3.

```
int FIPS_mode(void);
void *FIPS_rand_check(void);
```

B.3 Changed DES APIs in OpenSSL 0.9.7e

The following DES APIs have been changed in OpenSSL 0.9.7e and in HP SSL Version 1.3.

```
void DES_ecb3_encrypt(const unsigned char *input, unsigned char *output,
                          DES_key_schedule *ks1,DES_key_schedule *ks2,
                          DES_key_schedule *ks3, int enc);

void DES_ede3_cfb_encrypt(const unsigned char *in,unsigned char *out,
```

```
                                int numbits,long length,DES_key_schedule *ks1,
                                DES_key_schedule *ks2,DES_key_schedule *ks3,
                                DES_cblock *ivec,int enc);
```

B.4 New EVP APIs in OpenSSL 0.9.7e

The following EVP APIs are new in OpenSSL 0.9.7e and in HP SSL Version 1.3.

```
const EVP_CIPHER *EVP_des_cfb64(void);
const EVP_CIPHER *EVP_des_cfb1(void);
const EVP_CIPHER *EVP_des_cfb8(void);
const EVP_CIPHER *EVP_des_ede_cfb64(void);
const EVP_CIPHER *EVP_des_ede3_cfb64(void);
const EVP_CIPHER *EVP_des_ede3_cfb1(void);
const EVP_CIPHER *EVP_des_ede3_cfb8(void);

const EVP_CIPHER *EVP_idea_cfb64(void);
const EVP_CIPHER *EVP_rc2_cfb64(void);
const EVP_CIPHER *EVP_bf_cfb64(void);
const EVP_CIPHER *EVP_cast5_cfb64(void);
const EVP_CIPHER *EVP_rc5_32_12_16_cfb64(void

const EVP_CIPHER *EVP_aes_128_cfb1(void);
const EVP_CIPHER *EVP_aes_128_cfb8(void);
const EVP_CIPHER *EVP_aes_128_cfb128(void);
const EVP_CIPHER *EVP_aes_192_cfb1(void);
const EVP_CIPHER *EVP_aes_192_cfb8(void);
const EVP_CIPHER *EVP_aes_192_cfb128(void);
const EVP_CIPHER *EVP_aes_256_cfb1(void);
const EVP_CIPHER *EVP_aes_256_cfb8(void);
const EVP_CIPHER *EVP_aes_256_cfb128(void);
```

B.5 New SSL APIs in 0.9.7d

The following SSL APIs are new in OpenSSL 0.9.7d and in HP SSL Version 1.2.

```
KSSL_CTX        *kssl_ctx_free(KSSL_CTX *kssl_ctx);
KSSL_CTX        *kssl_ctx_new(void);
krb5_error_code kssl_cget_tkt(KSSL_CTX *kssl_ctx,  krb5_data **enc_tktp,
                              krb5_data *authenp, KSSL_ERR *kssl_err);
void            kssl_err_set(KSSL_ERR *kssl_err, int reason, char *text);
void            kssl_ctx_show(KSSL_CTX *kssl_ctx);
krb5_error_code kssl_validate_times(krb5_timestamp atime,
                                    krb5_ticket_times *ttimes); krb5_error_code
kssl_check_authent(KSSL_CTX *kssl_ctx, krb5_data *authentp,
                                  krb5_timestamp *atimep, KSSL_ERR *kssl_err); krb5_error_code
kssl_build_principal_2(krb5_context context,
                       krb5_principal *princ, int rlen, const char *realm,
                       int slen, const char *svc, int hlen, const char *host);
unsigned char   *kssl_skip_confound(krb5_enctype enctype, unsigned char *authn);
```

```
krb5_error_code kssl_sget_tkt(KSSL_CTX *kssl_ctx,  krb5_data *indata,
                              krb5_ticket_times *ttimes, KSSL_ERR *kssl_err); krb5_error_code
kssl_ctx_setkey(KSSL_CTX *kssl_ctx, krb5_keyblock *session); krb5_error_code
kssl_ctx_setprinc(KSSL_CTX *kssl_ctx, int which,
                                    krb5_data *realm, krb5_data *entity, int nentities);
void            kssl_krb5_free_data_contents(krb5_context context, krb5_data *data);
krb5_error_code kssl_ctx_setstring(KSSL_CTX *kssl_ctx, int which, char *text);


int  SSL_has_matching_session_id(const SSL *ssl, const unsigned char *id, unsigned int
id_len);
int  SSL_set_generate_session_id(SSL *, GEN_SESSION_CB);
int  SSL_CTX_set_generate_session_id(SSL_CTX *, GEN_SESSION_CB);
int  SSL_renegotiate_pending(SSL *s);
void SSL_CTX_set_msg_callback(SSL_CTX *ctx,
                        void (*cb)(int write_p, int version, int content_type,
                        const void *buf, size_t len, SSL *ssl, void *arg));
void SSL_set_msg_callback(SSL *ssl,
                        void (*cb)(int write_p, int version, int content_type,
                        const void *buf, size_t len, SSL *ssl, void *arg));
```

B.6 Changed SSL APIs in 0.9.7d

The following SSL APIs are have changed in OpenSSL 0.9.7d and in HP SSL Version 1.2. The information that has changed is underlined with ^^^^^.

```
0.9.7 - longSSL_ctrl(SSL *ssl,int cmd, long larg, void *parg);
0.9.6 - longSSL_ctrl(SSL *ssl,int cmd, long larg, char *parg);
                                                    ^^^^
0.9.7 - longSSL_CTX_ctrl(SSL_CTX *ctx,int cmd, long larg, void *parg);
0.9.6 - longSSL_CTX_ctrl(SSL_CTX *ctx,int cmd, long larg, char *parg);
                                                         ^^^^
0.9.7 - const char *SSL_alert_desc_string_long(int value);
0.9.6 -       char *SSL_alert_desc_string_long(int value);
        ^^^^^^^

0.9.7 - const char *SSL_alert_desc_string(int value);
0.9.6 -       char *SSL_alert_desc_string(int value);
        ^^^^^^^

0.9.7 - const char *SSL_alert_type_string_long(int value);
0.9.6 -       char *SSL_alert_type_string_long(int value);
        ^^^^^^^

0.9.7 - const char *SSL_alert_type_string(int value);
0.9.6 -       char *SSL_alert_type_string(int value);
        ^^^^^^^

0.9.7 - const char *SSL_rstate_string(const SSL *s);
0.9.6 -       char *SSL_rstate_string(SSL *s);
        ^^^^^^^                       ^^^^^^^^^

0.9.7 - const char *SSL_rstate_string_long(const SSL *s);
0.9.6 -       char *SSL_rstate_string_long(SSL *s);
```

Changed SSL APIs in 0.9.7d

```
        ^^^^^^^                              ^^^^^^^^^

0.9.7 - const char *SSL_state_string(const SSL *s);
0.9.6 -       char *SSL_state_string(SSL *s);
       ^^^^^^^^                      ^^^^^^^^^
0.9.7 - const char *SSL_state_string_long(const SSL *s);
0.9.6 -       char *SSL_state_string_long(SSL *s);
        ^^^^^^^                       ^^^^^^^^^

0.9.7 - void SSL_set_info_callback(SSL *ssl,void (*cb)
                                                (const SSL *ssl,int type,int val));
0.9.6 - void SSL_set_info_callback(SSL *ssl,void (*cb)());
                                                 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

0.9.7 - void (*SSL_get_info_callback(SSL *ssl))(const SSL *ssl,int type,int val);
0.9.6 - void (*SSL_get_info_callback(SSL *ssl))();
                                               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

C Open Source Notices

C.1 OpenSSL Open Source License

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).

C.2 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.]
```