Crypto Application

version 1.5

http://www.erlang.org

Typeset in LaTeX from SGML source using the DOCBUILDER 3.3.2 Document System.

Contents

Chapter 1

Crypto User's Guide

The *Crypto* application provides functions for computation of message digests, and functions for encryption and decryption.

This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/).

This product includes cryptographic software written by Eric Young (eay@cryptsoft.com).

This product includes software written by Tim Hudson (tjh@cryptsoft.com).

For full OpenSSL and SSLeay license texts, see Licenses [page 1].

1.1 Licenses

This chapter contains in extenso versions of the OpenSSL and SSLeay licenses.

1.1.1 OpenSSL License

1.1.2 SSLeay License

Crypto Reference Manual

Short Summaries

crypto

No functions are exported.

crypto

The following functions are exported:

- md5_mac(Key, Data) -> Mac
  [page 11] Compute an MD5 MAC message authentification code
- md5_mac_96(Key, Data) -> Mac
  [page 11] Compute an MD5 MAC message authentification code
- sha_mac(Key, Data) -> Mac
  [page 11] Compute an MD5 MAC message authentification code
- sha_mac_96(Key, Data) -> Mac
  [page 11] Compute an MD5 MAC message authentification code
- des_cbc_encrypt(Key, IVec, Text) -> Cipher
  [page 11] Encrypt Text according to DES in CBC mode
- des_cbc_decrypt(Key, IVec, Cipher) -> Text
  [page 11] Decrypt Cipher according to DES in CBC mode
- des3_cbc_encrypt(Key1, Key2, Key3, IVec, Text) -> Cipher
  [page 12] Encrypt Text according to DES3 in CBC mode
- des3_cbc_decrypt(Key1, Key2, Key3, IVec, Cipher) -> Text
  [page 12] Decrypt Cipher according to DES in CBC mode
- aes_cfb_128_encrypt(Key, IVec, Text) -> Cipher
  [page 12] Encrypt Text according to AES in Cipher Feedback mode or Cipher Block Chaining mode
- aes_cbc_128_encrypt(Key, IVec, Text) -> Cipher
  [page 12] Encrypt Text according to AES in Cipher Feedback mode or Cipher Block Chaining mode
- aes_cfb_128_decrypt(Key, IVec, Cipher) -> Text
  [page 12] Decrypt Cipher according to AES in Cipher Feedback mode or Cipher Block Chaining mode
- aes_cbc_128_decrypt(Key, IVec, Cipher) -> Text
  [page 12] Decrypt Cipher according to AES in Cipher Feedback mode or Cipher Block Chaining mode
- erlint(Mpint) ->
  [page 12] Convert between binary multi-precision integer and erlang big integer
- mpint(N) -> Mpint
  [page 12] Convert between binary multi-precision integer and erlang big integer
- rand_bytes(N) -> binary()
  [page 13] Generate a binary of random bytes
- rand_uniform(Lo, Hi) -> N
  [page 13] Generate a random number
- mod_exp(N, P, M) -> Result
  [page 13] Perform N ^ P mod M
- rsa_verify(Digest, Signature, Key) -> Verified
  [page 13] Verify the digest and signature using rsa with given public key.
- dss_verify(Digest, Signature, Key) -> Verified
  [page 13] Verify the digest and signature using rsa with given public key.

crypto

Application

The purpose of the Crypto application is to provide message digest and DES encryption for SMNPv3. It provides computation of message digests MD5 and SHA, and CBC-DES encryption and decryption.

Configuration

The following environment configuration parameters are defined for the Crypto application. Refer to application(3) for more information about configuration parameters.

`debug = true | false` <optional> Causes debug information to be written to standard error or standard output. Default is `false`.

OpenSSL libraries

The current implementation of the Erlang Crypto application is based on the *OpenSSL* package version 0.9.7 or higher. There are source and binary releases on the web.

Source releases of OpenSSL can be downloaded from the OpenSSL[1] project home page, or mirror sites listed there.

The same URL also contains links to some compiled binaries and libraries of OpenSSL (see the `Related/Binaries` menu) of which the Shining Light Productions Win32 and OpenSSL[2] pages are of interest for the Win32 user.

For some Unix flavours there are binary packages available on the net.

If you cannot find a suitable binary OpenSSL package, you have to fetch an OpenSSL source release and compile it.

You then have to compile and install the library `libcrypto.so` (Unix), or the library `libeay32.dll` (Win32).

For Unix The `crypto_drv` dynamic driver is delivered linked to OpenSSL libraries in `/usr/local/lib`, but the default dynamic linking will also accept libraries in `/lib` and `/usr/lib`.

If that is not applicable to the particular Unix operating system used, the example `Makefile` in the Crypto `priv/obj` directory, should be used as a basis for relinking the final version of the port program.

For `Win32` it is only required that the library can be found from the `PATH` environment variable, or that they reside in the appropriate `SYSTEM32` directory; hence no particular relinking is need. Hence no example `Makefile` for Win32 is provided.

[1] URL: `http://www.openssl.org`
[2] URL: `http://www.shininglightpro.com/search.php?searchname=Win32+OpenSSL`

SEE ALSO

application(3)

crypto

Erlang Module

This module provides a set of cryptographic functions.

References:

- md5: The MD5 Message Digest Algorithm (RFC 1321)
- sha: Secure Hash Standard (FIPS 180-2)
- hmac: Keyed-Hashing for Message Authentication (RFC 2104)
- des: Data Encryption Standard (FIPS 46-3)
- aes: Advanced Encryption Standard (AES) (FIPS 197)
- ecb, cbc, cfb, ofb: Recommendation for Block Cipher Modes of Operation (NIST SP 800-38A).
- rsa: Recommendation for Block Cipher Modes of Operation (NIST 800-38A)
- dss: Digital Signature Standard (FIPS 186-2)

The above publications can be found at NIST publications[3], at IETF[4].

Types

```
byte() = 0 ... 255
ioelem() = byte() | binary() | iolist()
iolist() = [ioelem()]
```

Exports

`start() -> ok`

Starts the crypto server.

`stop() -> ok`

Stops the crypto server.

`info() -> [atom()]`

Provides the available crypto functions in terms of a list of atoms.

`md5(Data) -> Digest`

Types:

[3] URL: `http://csrc.nist.gov/publications`
[4] URL: `http://www.ietf.org`

- Data = iolist() | binary()
- Digest = binary()

Computes an `MD5` message digest from `Data`, where the length of the digest is 128 bits (16 bytes).

`md5_init() -> Context`

Types:

- Context = binary()

Creates an MD5 context, to be used in subsequent calls to `md5_update/2`.

`md5_update(Context, Data) -> NewContext`

Types:

- Data = iolist() | binary()
- Context = NewContext = binary()

Updates an MD5 `Context` with `Data`, and returns a `NewContext`.

`md5_final(Context) -> Digest`

Types:

- Context = Digest = binary()

Finishes the update of an MD5 `Context` and returns the computed `MD5` message digest.

`sha(Data) -> Digest`

Types:

- Data = iolist() | binary()
- Digest = binary()

Computes an `SHA` message digest from `Data`, where the length of the digest is 160 bits (20 bytes).

`sha_init() -> Context`

Types:

- Context = binary()

Creates an SHA context, to be used in subsequent calls to `sha_update/2`.

`sha_update(Context, Data) -> NewContext`

Types:

- Data = iolist() | binary()
- Context = NewContext = binary()

Updates an SHA `Context` with `Data`, and returns a `NewContext`.

`sha_final(Context) -> Digest`

Types:

- Context = Digest = binary()

Finishes the update of an SHA `Context` and returns the computed `SHA` message digest.

`md5_mac(Key, Data) -> Mac`

Types:

- Key = Data = iolist() | binary()
- Mac = binary()

Computes an `MD5 MAC` message authentification code from `Key` and `Data`, where the the length of the Mac is 128 bits (16 bytes).

`md5_mac_96(Key, Data) -> Mac`

Types:

- Key = Data = iolist() | binary()
- Mac = binary()

Computes an `MD5 MAC` message authentification code from `Key` and `Data`, where the length of the Mac is 96 bits (12 bytes).

`sha_mac(Key, Data) -> Mac`

Types:

- Key = Data = iolist() | binary()
- Mac = binary()

Computes an `SHA MAC` message authentification code from `Key` and `Data`, where the length of the Mac is 160 bits (20 bytes).

`sha_mac_96(Key, Data) -> Mac`

Types:

- Key = Data = iolist() | binary()
- Mac = binary()

Computes an `SHA MAC` message authentification code from `Key` and `Data`, where the length of the Mac is 96 bits (12 bytes).

`des_cbc_encrypt(Key, IVec, Text) -> Cipher`

Types:

- Key = Text = iolist() | binary()
- IVec = Cipher = binary()

Encrypts `Text` according to DES in CBC mode. `Text` must be a multiple of 64 bits (8 bytes). `Key` is the DES key, and `IVec` is an arbitrary initializing vector. The lengths of `Key` and `IVec` must be 64 bits (8 bytes).

`des_cbc_decrypt(Key, IVec, Cipher) -> Text`

Types:

- Key = Cipher = iolist() | binary()
- IVec = Text = binary()

Decrypts `Cipher` according to DES in CBC mode. `Key` is the DES key, and `IVec` is an arbitrary initializing vector. `Key` and `IVec` must have the same values as those used when encrypting. `Cipher` must be a multiple of 64 bits (8 bytes). The lengths of `Key` and `IVec` must be 64 bits (8 bytes).

`des3_cbc_encrypt(Key1, Key2, Key3, IVec, Text) -> Cipher`

Types:

- Key1 =Key2 = Key3 Text = iolist() | binary()
- IVec = Cipher = binary()

Encrypts `Text` according to DES3 in CBC mode. `Text` must be a multiple of 64 bits (8 bytes). `Key1`, `Key2`, `Key3`, are the DES keys, and `IVec` is an arbitrary initializing vector. The lengths of each of `Key1`, `Key2`, `Key3` and `IVec` must be 64 bits (8 bytes).

`des3_cbc_decrypt(Key1, Key2, Key3, IVec, Cipher) -> Text`

Types:

- Key1 = Key2 = Key3 = Cipher = iolist() | binary()
- IVec = Text = binary()

Decrypts `Cipher` according to DES3 in CBC mode. `Key1`, `Key2`, `Key3` are the DES key, and `IVec` is an arbitrary initializing vector. `Key1`, `Key2`, `Key3` and `IVec` must and `IVec` must have the same values as those used when encrypting. `Cipher` must be a multiple of 64 bits (8 bytes). The lengths of `Key1`, `Key2`, `Key3`, and `IVec` must be 64 bits (8 bytes).

`aes_cfb_128_encrypt(Key, IVec, Text) -> Cipher`
`aes_cbc_128_encrypt(Key, IVec, Text) -> Cipher`

Types:

- Key = Text = iolist() | binary()
- IVec = Cipher = binary()

Encrypts `Text` according to AES in Cipher Feedback mode (CFB) or Cipher Block Chaining mode (CBC). `Text` must be a multiple of 128 bits (16 bytes). `Key` is the AES key, and `IVec` is an arbitrary initializing vector. The lengths of `Key` and `IVec` must be 128 bits (16 bytes).

`aes_cfb_128_decrypt(Key, IVec, Cipher) -> Text`
`aes_cbc_128_decrypt(Key, IVec, Cipher) -> Text`

Types:

- Key = Cipher = iolist() | binary()
- IVec = Text = binary()

Decrypts `Cipher` according to Cipher Feedback Mode (CFB) or Cipher Block Chaining mode (CBC). `Key` is the AES key, and `IVec` is an arbitrary initializing vector. `Key` and `IVec` must have the same values as those used when encrypting. `Cipher` must be a multiple of 128 bits (16 bytes). The lengths of `Key` and `IVec` must be 128 bits (16 bytes).

`erlint(Mpint) ->`
`mpint(N) -> Mpint`

Types:

- Mpint = binary()
- N = integer()

Convert a binary multi-precision integer `Mpint` to and from an erlang big integer. A multi-precision integer is a binary with the following form: `<<ByteLen:32/integer, Bytes:ByteLen/binary>>` where both `ByteLen` and `Bytes` are big-endian. Mpints are used in some of the functions in `crypto` and are not translated in the API for performance reasons.

`rand_bytes(N) -> binary()`

Types:

- N = integer()

Generates N bytes randomly uniform 0..255, and returns the result in a binary. Uses the `crypto` library pseudo-random number generator.

`rand_uniform(Lo, Hi) -> N`

Types:

- Lo, Hi, N = Mpint | integer()
- Mpint = binary()

Generate a random number `N, Lo =< N < Hi.` Uses the `crypto` library pseudo-random number generator. The arguments (and result) can be either erlang integers or binary multi-precision integers.

`mod_exp(N, P, M) -> Result`

Types:

- N, P, M, Result = Mpint
- Mpint = binary()

This function performs the exponentiation `N ^ P mod M`, using the `crypto` library.

`rsa_verify(Digest, Signature, Key) -> Verified`

Types:

- Verified = boolean()
- Digest, Signature = MPint
- Key = [E, N]
- E, N = MPint
- MPint = binary()

Verifies the digest and signature using the public key `Key`, using the `crypto` library function for RSA signature verification.

`dss_verify(Digest, Signature, Key) -> Verified`

Types:

- Verified = boolean()
- Digest, Signature = MPint
- Key = [P, Q, G, Y]

- P, Q, G, Y = MPint
- MPint = binary()

Verifies the digest and signature using the public key `Key`, using the `crypto` library function for DSS signature verification.

DES in CBC mode

The Data Encryption Standard (DES) defines an algoritm for encrypting and decrypting an 8 byte quantity using an 8 byte key (actually only 56 bits of the key is used).

When it comes to encrypting and decrypting blocks that are multiples of 8 bytes various modes are defined (NIST SP 800-38A). One of those modes is the Cipher Block Chaining (CBC) mode, where the encryption of an 8 byte segment depend not only of the contents of the segment itself, but also on the result of encrypting the previous segment: the encryption of the previous segment becomes the initializing vector of the encryption of the current segment.

Thus the encryption of every segment depends on the encryption key (which is secret) and the encryption of the previous segment, except the first segment which has to be provided with a first initializing vector. That vector could be chosen at random, or be counter of some kind. It does not have to be secret.

The following example is drawn from the old FIPS 81 standard (replaced by NIST SP 800-38A), where both the plain text and the resulting cipher text is settled. We use the Erlang bitsyntax to define binary literals. The following Erlang code fragment returns `true'.

```
Key = <<16#01,16#23,16#45,16#67,16#89,16#ab,16#cd,16#ef>>,
IVec = <<16#12,16#34,16#56,16#78,16#90,16#ab,16#cd,16#ef>>,
P = "Now is the time for all ",
C = crypto:des_cbc_encrypt(K, I, P),
C == <<16#e5,16#c7,16#cd,16#de,16#87,16#2b,16#f2,16#7c,
       16#43,16#e9,16#34,16#00,16#8c,16#38,16#9c,16#0f,
       16#68,16#37,16#88,16#49,16#9a,16#7c,16#05,16#f6>>,
<<"Now is the time for all ">> ==
                  crypto:des_cbc_decrypt(Key,IVec,C).
```

The following is true for the DES CBC mode. For all decompositions `P1 ++ P2 = P` of a plain text message `P` (where the length of all quantities are multiples of 8 bytes), the encryption `C` of `P` is equal to `C1 ++ C2`, where `C1` is obtained by encrypting `P1` with `Key` and the initializing vector `IVec`, and where `C2` is obtained by encrypting `P2` with `Key` and the initializing vector `l(C1)`, where `l(B)` denotes the last 8 bytes of the binary `B`.

Similarly, for all decompositions `C1 ++ C2 = C` of a cipher text message `C` (where the length of all quantities are multiples of 8 bytes), the decryption `P` of `C` is equal to `P1 ++ P2`, where `P1` is obtained by decrypting `C1` with `Key` and the initializing vector `IVec`, and where `P2` is obtained by decrypting `C2` with `Key` and the initializing vector `l(C1)`, where `l(.)` is as above.

For DES3 (which uses three 64 bit keys) the situation is the same.

Index of Modules and Functions

Modules are typed in *this way*.
Functions are typed in `this way`.