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_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 encryption

SYNOPSIS

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

The following functions have been deprecated since OpenSSL 3.0, and can be hidden entirely by defining **OPENSSL_API_COMPAT** with a suitable version value, see **openssl_user_macros**(7):

```
void DES_random_key(DES_cblock *ret);

int DES_set_key(const_DES_cblock *key, DES_key_schedule *schedule);
int DES_key_sched(const_DES_cblock *key, DES_key_schedule *schedule);
int DES_set_key_checked(const_DES_cblock *key, DES_key_schedule *schedule);
void DES_set_key_unchecked(const_DES_cblock *key, DES_key_schedule *schedule);

void DES_set_odd_parity(DES_cblock *key);
int DES_is_weak_key(const_DES_cblock *key);

void DES_ecb_encrypt(const_DES_cblock *input, DES_cblock *output,
                     DES_key_schedule *ks, int enc);
void DES_ecb2_encrypt(const_DES_cblock *input, DES_cblock *output,
                      DES_key_schedule *ks1, DES_key_schedule *ks2, int enc);
void DES_ecb3_encrypt(const_DES_cblock *input, DES_cblock *output,
                      DES_key_schedule *ks1, DES_key_schedule *ks2,
                      DES_key_schedule *ks3, int enc);

void DES_ncbc_encrypt(const unsigned char *input, unsigned char *output,
                      long length, DES_key_schedule *schedule, DES_cblock *ivec,
                      int enc);
void DES_cfb_encrypt(const unsigned char *in, unsigned char *out,
                     int numbits, long length, DES_key_schedule *schedule,
                     DES_cblock *ivec, int enc);
void DES_ofb_encrypt(const unsigned char *in, unsigned char *out,
```

```
                int numbits, long length, DES_key_schedule *schedule,
                DES_cblock *ivec);
    void DES_pcbc_encrypt(const unsigned char *input, unsigned char *output,
                 long length, DES_key_schedule *schedule, DES_cblock *ivec,
                 int enc);
    void DES_cfb64_encrypt(const unsigned char *in, unsigned char *out,
                 long length, DES_key_schedule *schedule, DES_cblock *ivec,
                 int *num, int enc);
    void DES_ofb64_encrypt(const unsigned char *in, unsigned char *out,
                 long length, DES_key_schedule *schedule, DES_cblock *ivec,
                 int *num);

    void DES_xcbc_encrypt(const unsigned char *input, unsigned char *output,
                 long length, DES_key_schedule *schedule, DES_cblock *ivec,
                 const_DES_cblock *inw, const_DES_cblock *outw, int enc);

    void DES_ede2_cbc_encrypt(const unsigned char *input, unsigned char *output,
                   long length, DES_key_schedule *ks1,
                   DES_key_schedule *ks2, DES_cblock *ivec, int enc);
    void DES_ede2_cfb64_encrypt(const unsigned char *in, unsigned char *out,
                    long length, DES_key_schedule *ks1,
                    DES_key_schedule *ks2, DES_cblock *ivec,
                    int *num, int enc);
    void DES_ede2_ofb64_encrypt(const unsigned char *in, unsigned char *out,
                    long length, DES_key_schedule *ks1,
                    DES_key_schedule *ks2, DES_cblock *ivec, int *num);

    void DES_ede3_cbc_encrypt(const unsigned char *input, unsigned char *output,
                   long length, DES_key_schedule *ks1,
                   DES_key_schedule *ks2, DES_key_schedule *ks3,
                   DES_cblock *ivec, int enc);
    void DES_ede3_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);
```

DESCRIPTION

All of the functions described on this page are deprecated. Applications should instead use **EVP_EncryptInit_ex**(3), **EVP_EncryptUpdate**(3) and **EVP_EncryptFinal_ex**(3) or the equivalently named decrypt functions.

This library contains a fast implementation of the DES encryption algorithm.

There are two phases to the use of DES encryption. The first is the generation of a *DES_key_schedule* from a key, the second is the actual encryption. A DES key is of type *DES_cblock*. This type consists of 8 bytes with odd parity. The least significant bit in each byte is the parity bit. The key schedule is an expanded form of the key; it is used to speed the encryption process.

DES_random_key() generates a random key. The random generator must be seeded when calling this function. If the automatic seeding or reseeding of the OpenSSL CSPRNG fails due to external circumstances (see **RAND**(7)), the operation will fail. If the function fails, 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 weak or semi-weak key. If the parity is wrong, then -1 is returned. If the key is a weak key, then -2 is returned. If an error is returned, the key schedule is not generated.

DES_set_key() works like **DES_set_key_checked()** and remains for backward compatibility.

DES_set_odd_parity() sets the parity of the passed *key* to odd.

DES_is_weak_key() returns 1 if the passed key is a weak key, 0 if it is ok.

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 nonzero (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 nonzero, 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 "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() encrypts/decrypts using the propagating cipher block chaining mode used by Kerberos v4. Its parameters are the same as **DES_ncbc_encrypt()**.

DES_cfb_encrypt() encrypts/decrypts using cipher feedback mode. This method takes an array of characters as input and outputs an 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 a small number of characters.

DES_cfb64_encrypt() implements CFB mode of DES with 64-bit feedback. Why is this useful you ask? Because this routine will allow you to encrypt an arbitrary number of bytes, without 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 an 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 a small number 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 mostly emulates the normal non-thread-safe semantics of **crypt**(3). The **salt** must be two ASCII characters.

The values returned by **DES_fcrypt()** and **DES_crypt()** are terminated by NUL character.

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.

BUGS

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()**.

NOTES

The **des** library was written to be source code compatible with the MIT Kerberos library.

Applications should use the higher level functions **EVP_EncryptInit**(3) etc. instead of calling these functions directly.

Single-key DES is insecure due to its short key size. ECB mode is not suitable for most applications; see **des_modes**(7).

RETURN VALUES

DES_set_key(), **DES_key_sched()**, and **DES_set_key_checked()** return 0 on success or negative values on error.

DES_is_weak_key() returns 1 if the passed key is a weak key, 0 if it is ok.

DES_cbc_cksum() and **DES_quad_cksum()** return 4-byte integer representing the last 4 bytes of the checksum of the input.

DES_fcrypt() returns a pointer to the caller-provided buffer and **DES_crypt()** - to a static buffer on success; otherwise they return NULL.

SEE ALSO

des_modes(7), **EVP_EncryptInit**(3)

HISTORY

All of these functions were deprecated in OpenSSL 3.0.

The requirement that the **salt** parameter to **DES_crypt()** and **DES_fcrypt()** be two ASCII characters was first enforced in OpenSSL 1.1.0. Previous versions tried to use the letter uppercase **A** if both character were not present, and could crash when given non-ASCII on some platforms.

COPYRIGHT

Copyright 2000-2021 The OpenSSL Project Authors. All Rights Reserved.

Licensed under the Apache License 2.0 (the "License"). You may not use this file except in compliance with the License. You can obtain a copy in the file LICENSE in the source distribution or at <https://www.openssl.org/source/license.html>.