## NAME

**mhash - Hash Library**

## VERSION

mhash 0.9.2

## SYNOPSIS

```
#include "mhash.h"
```

**Informative Functions**

```
size_t   mhash_count(void);
size_t   mhash_get_block_size(hashid type);
char    *mhash_get_hash_name(hashid type);
size_t   mhash_get_hash_pblock(hashid type);
hashid   mhash_get_mhash_algo( MHASH);
```

**Key Generation Functions**

```
int      mhash_keygen_ext(keygenid algorithm, KEYGEN algorithm_data,
              void* keyword, int keysize,
              unsigned char* password, int passwordlen);
```

**Initializing Functions**

```
MHASH    mhash_init(hashid type);
MHASH    mhash_hmac_init(const hashid type, void *key, int keysize, int block);
MHASH    mhash_cp( MHASH);
```

**Update Functions**

```
int      mhash(MHASH thread, const void *plaintext, size_t size);
```

**Save/Restore Functions**

```
int      mhash_save_state_mem(MHASH thread, void *mem, int* mem_size );
MHASH    mhash_restore_state_mem(void* mem);
```

**Finalizing Functions**

```
void    mhash_deinit(MHASH thread, void *result);
void    *mhash_end(MHASH thread);
void    *mhash_end_m(MHASH thread, void* (*hash_malloc)(size_t));

void    *mhash_hmac_end(MHASH thread);
void    *mhash_hmac_end_m(MHASH thread, void* (*hash_malloc)(size_t));
int     mhash_hmac_deinit(MHASH thread, void *result);
```

**Available Hashes**

*CRC32*: The crc32 algorithm is used to compute checksums. The two variants used in mhash are: **MHASH_CRC32** (like the one used in ethernet) and **MHASH_CRC32B** (like the one used in ZIP programs).

*ADLER32*: The adler32 algorithm is used to compute checksums. It is faster than CRC32 and it is considered to be as reliable as CRC32. This algorithm is defined as **MHASH_ADLER32**.

*MD5*: The MD5 algorithm by Ron Rivest and RSA. In mhash this algorithm is defined as **MHASH_MD5**.

*MD4*: The MD4 algorithm by Ron Rivest and RSA. This algorithm is considered broken, so don’t use it. In mhash this algorithm is defined as **MHASH_MD4**.

*SHA1*/*SHA256*: The SHA algorithm by US. NIST/NSA. This algorithm is specified for use in the NIST’s Digital Signature Standard. In mhash these algorithm are defined as **MHASH_SHA1** and **MHASH_SHA256**.

*HAVAL*: HAVAL is a one-way hashing algorithm with variable length of output. HAVAL is a modification of MD5. Defined in mhash as: **MHASH_HAVAL256, MHASH_HAVAL192, MHASH_HAVAL160, MHASH_HAVAL128**.

*RIPEMD160*: RIPEMD-160 is a 160-bit cryptographic hash function, designed by Hans Dobbertin, Antoon Bosselaers, and Bart Preneel. It is intended to be used as a secure replacement for the 128-bit hash functions MD4, MD5, and RIPEMD. MD4 and MD5 were developed by Ron Rivest for RSA Data Security, while RIPEMD was developed in the framework of the EU project RIPE (RACE Integrity Primitives Evaluation, 1988-1992). In mhash this algorithm is defined as **MHASH_RIPEMD160**.

*TIGER*: Tiger is a fast hash function, by Eli Biham and Ross Anderson. Tiger was designed to be very fast on modern computers, and in particular on the state-of-the-art 64-bit computers, while it is still not slower than other suggested hash functions on 32-bit machines. In mhash this algorithm is defined as: **MHASH_TIGER, MHASH_TIGER160, MHASH_TIGER128**.

*GOST*: GOST algorithm is a russian standard and it uses the GOST encryption algorithm to produce a 256 bit hash value. This algorithm is specified for use in the Russian Digital Signature Standard. In mhash this algorithm is defined as **MHASH_GOST**.

### Available Key Generation algorithms

*KEYGEN_MCRYPT*: The key generator used in mcrypt.

*KEYGEN_ASIS*: Just returns the password as binary key.

*KEYGEN_HEX*: Just converts a hex key into a binary one.

*KEYGEN_PKDES*: The transformation used in Phil Karn's DES encryption program.

*KEYGEN_S2K_SIMPLE*: The OpenPGP (rfc2440) Simple S2K.

*KEYGEN_S2K_SALTED*: The OpenPGP Salted S2K.

*KEYGEN_S2K_ISALTED*: The OpenPGP Iterated Salted S2K.

## DESCRIPTION

The **mhash** library provides an easy to use C interface for several *hash algorithms* (also known as "one-way" algorithms). These can be used to create checksums, message digests and more. Currently, MD5, SHA1, GOST, TIGER, RIPE-MD160, HAVAL and several other algorithms are supported. **mhash** support *HMAC generation* (a mechanism for message authentication using cryptographic hash functions, and is described in rfc2104). HMAC can be used to create message digests using a secret key, so that these message digests cannot be regenerated (or replaced) by someone else. A key generation mechanism was added to **mhash** since *key generation* algorithms usually involve hash algorithms.

## API FUNCTIONS

We will describe the API of **mhash** in detail now. The order follows the one in the SYNOPSIS directly.

size_t **mhash_count**(void);

This returns the "hashid" of the last available hash. Hashes are numbered from 0 to "mhash_count()".

size_t **mhash_get_block_size**(hashid *type*);
If *type* exists, this returns the used blocksize of the hash *type* in bytes. Otherwise, it returns 0.

char ***mhash_get_hash_name**(hashid *type*);
If *type* exists, this returns the name of the hash *type*. Otherwise, a "NULL" pointer is returned. The string is allocated with *malloc*(3) seperately, so do not forget to *free*(3) it.

const char ***mhash_get_hash_name_static**(hashid *type*);
If *type* exists, this returns the name of the hash *type*. Otherwise, a "NULL" pointer is returned.

size_t **mhash_get_hash_pblock**(hashid *type*);
It returns the block size that the algorithm operates. This is used in mhash_hmac_init. If the return value is 0 you shouldn't use that algorithm in HMAC.

hashid **mhash_get_mhash_algo**(MHASH *src*);
Returns the algorithm used in the state of *src*.

MHASH **mhash_init**(hashid *type*);
This setups a context to begin hashing using the algorithm *type*. It returns a descriptor to that context which will result in leaking memory, if you do not call *mhash_deinit*(3) later. Returns "MHASH_FAILED" on failure.

MHASH **mhash_hmac_init**(const hashid *type*, void **key*, int *keysize*, int *block*);
This setups a context to begin hashing using the algorithm type in HMAC mode. *key* should be a pointer to the key and *keysize* its len. The *block* is the block size (in bytes) that the algorithm operates. It should be obtained by *mhash_get_hash_pblock()*. If its 0 it defaults to 64. After calling it you should use *mhash()* to update the context. It returns a descriptor to that context which will result in leaking memory, if you do not call *mhash_hmac_deinit*(3) later. Returns "MHASH_FAILED" on failure.

MHASH **mhash_cp**(MHASH *src*);
This setups a new context using the state of *src*.

int **mhash**(MHASH *thread*, const void **plaintext*, size_t *size*);
This updates the context described by *thread* with *plaintext*. *size* is the length of *plaintext* which may be binary data.

int **mhash_save_state_mem**( MHASH *thread*, void **mem*, int* *mem_size*);
Saves the state of a hashing algorithm such that it can be restored at some later point in time using **mhash_restore_state_mem**(). *mem_size* should contain the size of the given *mem* pointer. If it is

not enough to hold the buffer the required value will be copied there.

MHASH **mhash_restore_state_mem**(void* *mem*);

Restores the state of a hashing algorithm that was saved using **mhash_save_state_mem**(). Use like **mhash_init**().

void ***mhash_end**(MHASH *thread*);

This frees all resources associated with *thread* and returns the result of the whole hashing operation (the ‘‘*digest*’’).

void **mhash_deinit**(MHASH *thread*, void* digest);

This frees all resources associated with *thread* and stores the result of the whole hashing operation in memory pointed by *digest*. *digest* may be null.

void ***mhash_hmac_end**(MHASH *thread*);

This frees all resources associated with thread and returns the result of the whole hashing operation (the ‘‘*mac*’’).

int **mhash_hmac_deinit**(MHASH *thread*, void* digest);

This frees all resources associated with *thread* and stores the result of the whole hashing operation in memory pointed by digest. Digest may be null. Returns non-zero in case of an error.

void ***mhash_end_m**(MHASH *thread*, void* (*hash_malloc)(size_t));

This frees all resources associated with *thread* and returns the result of the whole hashing operation (the ‘‘*digest*’’). The result will be allocated by using the *hash_malloc()* function provided.

void ***mhash_hmac_end**(MHASH *thread*, void* (*hash_malloc)(size_t));

This frees all resources associated with thread and returns the result of the whole hashing operation (the ‘‘*mac*’’). The result will be allocated by using the *hash_malloc()* function provided.

## KEYGEN API FUNCTIONS

We will now describe the Key Generation API of **mhash** in detail.

int **mhash_keygen_ext**(keygenid *algorithm*, KEYGEN *algorithm_data*, void* *keyword*, int *keysize*, unsigned char* *password*, int *passwordlen*);

This function, generates a key from a password. The password is read from *password* and it’s len should be in *passwordlen*. The key generation algorithm is specified in *algorithm*, and that algorithm may (internally) use the KEYGEN structure. The KEYGEN structure consists of:

```
 typedef struct keygen {
        hashid          hash_algorithm[2];
        unsigned int    count;
        void*           salt;
        int             salt_size;
    } KEYGEN;
```

The algorithm(s) specified in *algorithm_data.hash_algorithm*, should be hash algorithms and may be used by the key generation algorithm. Some key generation algorithms may use more than one hash algorithms (view also *mhash_keygen_uses_hash_algorithm()*). If it is desirable (and supported by the algorithm, eg. KEYGEN_S2K_SALTED) a salt may be specified in *algorithm_data.salt* of size *algorithm_data.salt_size* or may be NULL.

The algorithm may use the *algorithm_data.count* internally (eg. KEYGEN_S2K_ISALTED). The generated keyword is stored in *keyword*, which should be (at least) *keysize* bytes long. The generated keyword is a binary one. Returns a negative number on failure.

int **mhash_keygen_uses_salt**( keygenid *algorithm*);
This function returns 1 if the specified key generation algorithm needs a salt to be specified.

int **mhash_keygen_uses_count**( keygenid *algorithm*);
This function returns 1 if the specified key generation algorithm needs the algorithm_data.count field in *mhash_keygen_ext()*. The count field tells the algorithm to hash repeatedly the password and to stop when **count** bytes have been processed.

int **mhash_get_keygen_salt_size**( keygenid *algorithm*);
This function returns the size of the salt size, that the specific *algorithm* will use. If it returns 0, then there is no limitation in the size.

int **mhash_get_keygen_max_key_size**( keygenid *algorithm*);
This function returns the maximum size of the key, that the key generation algorithm may produce. If it returns 0, then there is no limitation in the size.

int **mhash_keygen_uses_hash_algorithm**( keygenid *algorithm*);
This function returns the number of the hash algorithms the key generation algorithm will use. If it is 0 then no hash algorithm is used by the key generation algorithm. This is for the *algorithm_data.hash_algorithm* field in *mhash_keygen_ext()*. If

size_t **mhash_keygen_count**(void);
This returns the "keygenid" of the last available key generation algorithm. Algorithms are numbered from 0 to "mhash_keygen_count()".

char ***mhash_get_keygen_name**(keygenid *type*);
If *type* exists, this returns the name of the keygen *type*. Otherwise, a "NULL" pointer is returned. The string is allocated with *malloc*(3) seperately, so do not forget to *free*(3) it.

const char ***mhash_get_keygen_name_static**(keygenid *type*);
If *type* exists, this returns the name of the keygen *type*. Otherwise, a "NULL" pointer is returned.

## EXAMPLE

Hashing STDIN until EOF.

```
#include <mhash.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
        int i;
        MHASH td;
        unsigned char buffer;
        unsigned char hash[16]; /* enough size for MD5 */

        td = mhash_init(MHASH_MD5);

        if (td == MHASH_FAILED) exit(1);

        while (fread(&buffer, 1, 1, stdin) == 1) {
                mhash(td, &buffer, 1);
        }

        mhash_deinit(td, hash);

        printf("Hash:");
        for (i = 0; i < mhash_get_block_size(MHASH_MD5); i++) {
                printf("%.2x", hash[i]);
        }
        printf("\n");

        exit(0);
}
```

## EXAMPLE

An example program using HMAC:

```
#include <mhash.h>
#include <stdio.h>

int main()
{

        char password[] = "Jefe";
        int keylen = 4;
        char data[] = "what do ya want for nothing?";
        int datalen = 28;
        MHASH td;
        unsigned char mac[16];
        int j;

        td = mhash_hmac_init(MHASH_MD5, password, keylen,
                             mhash_get_hash_pblock(MHASH_MD5));

        mhash(td, data, datalen);
        mhash_hmac_deinit(td, mac);

/*
 * The output should be 0x750c783e6ab0b503eaa86e310a5db738
 * according to RFC 2104.
 */

        printf("0x");
        for (j = 0; j < mhash_get_block_size(MHASH_MD5); j++) {
                printf("%.2x", mac[j]);
        }
        printf("\n");

        exit(0);
}
```

## HISTORY

This library was originally written by *Nikos Mavroyanopoulos* <nmav@hellug.gr> who passed the project over to *Sascha Schumann* <sascha@schumann.cx> in May 1999. Sascha maintained it until

March 2000. The library is now maintained by *Nikos Mavroyanopoulos*.

## BUGS

If you find any, please send a bug report (preferrably together with a patch) to the maintainer with a detailed description on how to reproduce the bug.

## AUTHORS

Sascha Schumann <sascha@schumann.cx> Nikos Mavroyanopoulos <nmav@hellug.gr>