Nettle Manual

For the Nettle Library version 2.0

Niels Möller

This manual is for the Nettle library (version 2.0), a low-level cryptographic library.

Originally written 2001 by Niels Möller, updated 2009.

Table of Contents

1 Introduction

Nettle is a cryptographic library that is designed to fit easily in more or less any context: In crypto toolkits for object-oriented languages (C++, Python, Pike, ...), in applications like LSH or GNUPG, or even in kernel space. In most contexts, you need more than the basic cryptographic algorithms, you also need some way to keep track of available algorithms, their properties and variants. You often have some algorithm selection process, often dictated by a protocol you want to implement.

And as the requirements of applications differ in subtle and not so subtle ways, an API that fits one application well can be a pain to use in a different context. And that is why there are so many different cryptographic libraries around.

Nettle tries to avoid this problem by doing one thing, the low-level crypto stuff, and providing a *simple* but general interface to it. In particular, Nettle doesn't do algorithm selection. It doesn't do memory allocation. It doesn't do any I/O.

The idea is that one can build several application and context specific interfaces on top of Nettle, and share the code, test cases, benchmarks, documentation, etc. Examples are the Nettle module for the Pike language, and LSH, which both use an object-oriented abstraction on top of the library.

This manual explains how to use the Nettle library. It also tries to provide some background on the cryptography, and advice on how to best put it to use.

2 Copyright

Nettle is distributed under the GNU General Public License (GPL) (see the file COPYING for details). However, most of the individual files are dual licensed under less restrictive licenses like the GNU Lesser General Public License (LGPL), or are in the public domain. This means that if you don't use the parts of nettle that are GPL-only, you have the option to use the Nettle library just as if it were licensed under the LGPL. To find the current status of particular files, you have to read the copyright notices at the top of the files.

This manual is in the public domain. You may freely copy it in whole or in part, e.g., into documentation of programs that build on Nettle. Attribution, as well as contribution of improvements to the text, is of course appreciated, but it is not required.

A list of the supported algorithms, their origins and licenses:

AES The implementation of the AES cipher (also known as rijndael) is written by Rafael Sevilla. Assembler for x86 by Rafael Sevilla and Niels Möller, Sparc assembler by Niels Möller. Released under the LGPL.

ARCFOUR
The implementation of the ARCFOUR (also known as RC4) cipher is written by Niels Möller. Released under the LGPL.

ARCTWO
The implementation of the ARCTWO (also known as RC2) cipher is written by Nikos Mavroyanopoulos and modified by Werner Koch and Simon Josefsson. Released under the LGPL.

BLOWFISH
The implementation of the BLOWFISH cipher is written by Werner Koch, copyright owned by the Free Software Foundation. Also hacked by Ray Dassen and Niels Möller. Released under the GPL.

CAST128 The implementation of the CAST128 cipher is written by Steve Reid. Released into the public domain.

DES The implementation of the DES cipher is written by Dana L. How, and released under the LGPL.

MD2 The implementation of MD2 is written by Andrew Kuchling, and hacked some by Andreas Sigfridsson and Niels Möller. Python Cryptography Toolkit license (essentially public domain).

MD4 This is almost the same code as for MD5 below, with modifications by Marcus Comstedt. Released into the public domain.

MD5 The implementation of the MD5 message digest is written by Colin Plumb. It has been hacked some more by Andrew Kuchling and Niels Möller. Released into the public domain.

SERPENT
The implementation of the SERPENT cipher is written by Ross Anderson, Eli Biham, and Lars Knudsen, adapted to LSH by Rafael Sevilla, and to Nettle by Niels Möller. Released under the GPL.

SHA1 The C implementation of the SHA1 message digest is written by Peter Gutmann, and hacked some more by Andrew Kuchling and Niels Möller. Released into the public domain. Assembler for x86 by Niels Möller, released under the LGPL.

SHA256 Written by Niels Möller, using Peter Gutmann's SHA1 code as a model. Released under the LGPL.

TWOFISH
The implementation of the TWOFISH cipher is written by Ruud de Rooij. Released under the LGPL.

RSA Written by Niels Möller, released under the LGPL. Uses the GMP library for bignum operations.

DSA Written by Niels Möller, released under the LGPL. Uses the GMP library for bignum operations.

3 Conventions

For each supported algorithm, there is an include file that defines a *context struct*, a few constants, and declares functions for operating on the context. The context struct encapsulates all information needed by the algorithm, and it can be copied or moved in memory with no unexpected effects.

For consistency, functions for different algorithms are very similar, but there are some differences, for instance reflecting if the key setup or encryption function differ for encryption and decryption, and whether or not key setup can fail. There are also differences between algorithms that don't show in function prototypes, but which the application must nevertheless be aware of. There is no big difference between the functions for stream ciphers and for block ciphers, although they should be used quite differently by the application.

If your application uses more than one algorithm, you should probably create an interface that is tailor-made for your needs, and then write a few lines of glue code on top of Nettle.

By convention, for an algorithm named `foo`, the struct tag for the context struct is `foo_ctx`, constants and functions uses prefixes like `FOO_BLOCK_SIZE` (a constant) and `foo_set_key` (a function).

In all functions, strings are represented with an explicit length, of type `unsigned`, and a pointer of type `uint8_t *` or `const uint8_t *`. For functions that transform one string to another, the argument order is length, destination pointer and source pointer. Source and destination areas are of the same length. Source and destination may be the same, so that you can process strings in place, but they *must not* overlap in any other way.

4 Example

A simple example program that reads a file from standard input and writes its SHA1 checksum on standard output should give the flavor of Nettle.

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

#include <nettle/sha.h>

#define BUF_SIZE 1000

static void
display_hex(unsigned length, uint8_t *data)
{
  unsigned i;

  for (i = 0; i<length; i++)
    printf("%02x ", data[i]);

  printf("\n");
}

int
main(int argc, char **argv)
{
  struct sha1_ctx ctx;
  uint8_t buffer[BUF_SIZE];
  uint8_t digest[SHA1_DIGEST_SIZE];

  sha1_init(&ctx);
  for (;;)
  {
    int done = fread(buffer, 1, sizeof(buffer), stdin);
    sha1_update(&ctx, done, buffer);
    if (done < sizeof(buffer))
      break;
  }
  if (ferror(stdin))
    return EXIT_FAILURE;

  sha1_digest(&ctx, SHA1_DIGEST_SIZE, digest);

  display_hex(SHA1_DIGEST_SIZE, digest);
  return EXIT_SUCCESS;
}
```

On a typical Unix system, this program can be compiled and linked with the command line

```
cc sha-example.c -o sha-example -lnettle
```

5 Linking

Nettle actually consists of two libraries, '`libnettle`' and '`libhogweed`'. The '`libhogweed`' library contains those functions of Nettle that uses bignum operations, and depends on the GMP library. With this division, linking works the same for both static and dynamic libraries.

If an application uses only the symmetric crypto algorithms of Nettle (i.e., block ciphers, hash functions, and the like), it's sufficient to link with `-lnettle`. If an application also uses public-key algorithms, it must be linked with `-lhogweed -lnettle -lgmp`.

6 Reference

This chapter describes all the Nettle functions, grouped by family.

6.1 Hash functions

A cryptographic *hash function* is a function that takes variable size strings, and maps them to strings of fixed, short, length. There are naturally lots of collisions, as there are more possible 1MB files than 20 byte strings. But the function is constructed such that is hard to find the collisions. More precisely, a cryptographic hash function `H` should have the following properties:

One-way Given a hash value `H(x)` it is hard to find a string `x` that hashes to that value.

Collision-resistant
It is hard to find two different strings, `x` and `y`, such that `H(x) = H(y)`.

Hash functions are useful as building blocks for digital signatures, message authentication codes, pseudo random generators, association of unique id:s to documents, and many other things.

The most commonly used hash functions are MD5 and SHA1. Unfortunately, both these fail the collision-resistance requirement; cryptologists have found ways to construct colliding inputs. The recommended hash function for new applications is SHA256, even though it uses a structure similar to MD5 and SHA1. Constructing better hash functions is an urgent research problem.

6.1.1 MD5

MD5 is a message digest function constructed by Ronald Rivest, and described in *RFC 1321*. It outputs message digests of 128 bits, or 16 octets. Nettle defines MD5 in '`<nettle/md5.h>`'.

`struct md5_ctx` [Context struct]

`MD5_DIGEST_SIZE` [Constant]
The size of an MD5 digest, i.e. 16.

`MD5_DATA_SIZE` [Constant]
The internal block size of MD5. Useful for some special constructions, in particular HMAC-MD5.

`void md5_init` (*struct md5_ctx* ***ctx***) [Function]
Initialize the MD5 state.

`void md5_update` (*struct md5_ctx* ***ctx***, *unsigned* ***length***, *const uint8_t* ***data***) [Function]
Hash some more data.

`void md5_digest` (*struct md5_ctx* ***ctx***, *unsigned* ***length***, *uint8_t* ***digest***) [Function]
Performs final processing and extracts the message digest, writing it to *digest*. *length* may be smaller than `MD5_DIGEST_SIZE`, in which case only the first *length* octets of the digest are written.

This function also resets the context in the same way as `md5_init`.

The normal way to use MD5 is to call the functions in order: First `md5_init`, then `md5_update` zero or more times, and finally `md5_digest`. After `md5_digest`, the context is reset to its initial state, so you can start over calling `md5_update` to hash new data.

To start over, you can call `md5_init` at any time.

6.1.2 MD2

MD2 is another hash function of Ronald Rivest's, described in *RFC 1319*. It outputs message digests of 128 bits, or 16 octets. Nettle defines MD2 in '`<nettle/md2.h>`'.

`struct md2_ctx` [Context struct]

`MD2_DIGEST_SIZE` [Constant]
The size of an MD2 digest, i.e. 16.

`MD2_DATA_SIZE` [Constant]
The internal block size of MD2.

`void md2_init` (*struct md2_ctx* ***ctx**) [Function]
Initialize the MD2 state.

`void md2_update` (*struct md2_ctx* ***ctx**, *unsigned* **length**, *const uint8_t* ***data**) [Function]
Hash some more data.

`void md2_digest` (*struct md2_ctx* ***ctx**, *unsigned* **length**, *uint8_t* ***digest**) [Function]
Performs final processing and extracts the message digest, writing it to *digest*. *length* may be smaller than `MD2_DIGEST_SIZE`, in which case only the first *length* octets of the digest are written.

This function also resets the context in the same way as `md2_init`.

6.1.3 MD4

MD4 is a predecessor of MD5, described in *RFC 1320*. Like MD5, it is constructed by Ronald Rivest. It outputs message digests of 128 bits, or 16 octets. Nettle defines MD4 in '`<nettle/md4.h>`'. Use of MD4 is not recommended, but it is sometimes needed for compatibility with existing applications and protocols.

`struct md4_ctx` [Context struct]

`MD4_DIGEST_SIZE` [Constant]
The size of an MD4 digest, i.e. 16.

`MD4_DATA_SIZE` [Constant]
The internal block size of MD4.

`void md4_init` (*struct md4_ctx* ***ctx**) [Function]
Initialize the MD4 state.

`void md4_update` (*struct md4_ctx* ***ctx**, *unsigned* **length**, *const uint8_t* ***data**) [Function]
Hash some more data.

`void md4_digest` (*struct md4_ctx* ***ctx***, *unsigned* ***length***, *uint8_t* ***digest***) [Function]

Performs final processing and extracts the message digest, writing it to *digest*. *length* may be smaller than `MD4_DIGEST_SIZE`, in which case only the first *length* octets of the digest are written.

This function also resets the context in the same way as `md4_init`.

6.1.4 SHA1

SHA1 is a hash function specified by *NIST* (The U.S. National Institute for Standards and Technology). It outputs hash values of 160 bits, or 20 octets. Nettle defines SHA1 in '`<nettle/sha.h>`'.

The functions are analogous to the MD5 ones.

`struct sha1_ctx` [Context struct]

`SHA1_DIGEST_SIZE` [Constant]

The size of an SHA1 digest, i.e. 20.

`SHA1_DATA_SIZE` [Constant]

The internal block size of SHA1. Useful for some special constructions, in particular HMAC-SHA1.

`void sha1_init` (*struct sha1_ctx* ***ctx***) [Function]

Initialize the SHA1 state.

`void sha1_update` (*struct sha1_ctx* ***ctx***, *unsigned* ***length***, *const uint8_t* ***data***) [Function]

Hash some more data.

`void sha1_digest` (*struct sha1_ctx* ***ctx***, *unsigned* ***length***, *uint8_t* ***digest***) [Function]

Performs final processing and extracts the message digest, writing it to *digest*. *length* may be smaller than `SHA1_DIGEST_SIZE`, in which case only the first *length* octets of the digest are written.

This function also resets the context in the same way as `sha1_init`.

6.1.5 SHA256

SHA256 is another hash function specified by *NIST*, intended as a replacement for SHA1, generating larger digests. It outputs hash values of 256 bits, or 32 octets. Nettle defines SHA256 in '`<nettle/sha.h>`'.

The functions are analogous to the MD5 ones.

`struct sha256_ctx` [Context struct]

`SHA256_DIGEST_SIZE` [Constant]

The size of an SHA256 digest, i.e. 32.

`SHA256_DATA_SIZE` [Constant]

The internal block size of SHA256. Useful for some special constructions, in particular HMAC-SHA256.

`void sha256_init` (*struct sha256_ctx* *`ctx`) [Function]
Initialize the SHA256 state.

`void sha256_update` (*struct sha256_ctx* *`ctx`, *unsigned* `length`, *const uint8_t* *`data`) [Function]
Hash some more data.

`void sha256_digest` (*struct sha256_ctx* *`ctx`, *unsigned* `length`, *uint8_t* *`digest`) [Function]
Performs final processing and extracts the message digest, writing it to *digest*. *length* may be smaller than `SHA256_DIGEST_SIZE`, in which case only the first *length* octets of the digest are written.

This function also resets the context in the same way as `sha256_init`.

6.1.6 `struct nettle_hash`

Nettle includes a struct including information about the supported hash functions. It is defined in '`<nettle/nettle-meta.h>`', and is used by Nettle's implementation of HMAC see Section 6.4 [Keyed hash functions], page 23.

`struct nettle_hash` *name context_size digest_size block_size init update digest* [Meta struct]
The last three attributes are function pointers, of types `nettle_hash_init_func`, `nettle_hash_update_func`, and `nettle_hash_digest_func`. The first argument to these functions is `void *` pointer so a context struct, which is of size `context_size`.

`struct nettle_cipher nettle_md2` [Constant Struct]
`struct nettle_cipher nettle_md4` [Constant Struct]
`struct nettle_cipher nettle_md5` [Constant Struct]
`struct nettle_cipher nettle_sha1` [Constant Struct]
`struct nettle_cipher nettle_sha256` [Constant Struct]
These are all the hash functions that Nettle implements.

6.2 Cipher functions

A *cipher* is a function that takes a message or *plaintext* and a secret *key* and transforms it to a *ciphertext*. Given only the ciphertext, but not the key, it should be hard to find the plaintext. Given matching pairs of plaintext and ciphertext, it should be hard to find the key.

There are two main classes of ciphers: Block ciphers and stream ciphers.

A block cipher can process data only in fixed size chunks, called *blocks*. Typical block sizes are 8 or 16 octets. To encrypt arbitrary messages, you usually have to pad it to an integral number of blocks, split it into blocks, and then process each block. The simplest way is to process one block at a time, independent of each other. That mode of operation is called *ECB*, Electronic Code Book mode. However, using ECB is usually a bad idea. For a start, plaintext blocks that are equal are transformed to ciphertext blocks that are equal; that leaks information about the plaintext. Usually you should apply the cipher is some "feedback mode", *CBC* (Cipher Block Chaining) and *CTR* (Counter mode) being two of

of the most popular. See See Section 6.3 [Cipher modes], page 20, for information on how to apply CBC and CTR with Nettle.

A stream cipher can be used for messages of arbitrary length. A typical stream cipher is a keyed pseudo-random generator. To encrypt a plaintext message of n octets, you key the generator, generate n octets of pseudo-random data, and XOR it with the plaintext. To decrypt, regenerate the same stream using the key, XOR it to the ciphertext, and the plaintext is recovered.

Caution: The first rule for this kind of cipher is the same as for a One Time Pad: *never* ever use the same key twice.

A common misconception is that encryption, by itself, implies authentication. Say that you and a friend share a secret key, and you receive an encrypted message. You apply the key, and get a plaintext message that makes sense to you. Can you then be sure that it really was your friend that wrote the message you're reading? The answer is no. For example, if you were using a block cipher in ECB mode, an attacker may pick up the message on its way, and reorder, delete or repeat some of the blocks. Even if the attacker can't decrypt the message, he can change it so that you are not reading the same message as your friend wrote. If you are using a block cipher in CBC mode rather than ECB, or are using a stream cipher, the possibilities for this sort of attack are different, but the attacker can still make predictable changes to the message.

It is recommended to *always* use an authentication mechanism in addition to encrypting the messages. Popular choices are Message Authentication Codes like HMAC-SHA1 see Section 6.4 [Keyed hash functions], page 23, or digital signatures like RSA.

Some ciphers have so called "weak keys", keys that results in undesirable structure after the key setup processing, and should be avoided. In Nettle, the presence of weak keys for a cipher mean that the key setup function can fail, so you have to check its return value. In addition, the context struct has a field `status`, that is set to a non-zero value if key setup fails. When possible, avoid algorithm that have weak keys. There are several good ciphers that don't have any weak keys.

To encrypt a message, you first initialize a cipher context for encryption or decryption with a particular key. You then use the context to process plaintext or ciphertext messages. The initialization is known as called *key setup*. With Nettle, it is recommended to use each context struct for only one direction, even if some of the ciphers use a single key setup function that can be used for both encryption and decryption.

6.2.1 AES

AES is a block cipher, specified by NIST as a replacement for the older DES standard. The standard is the result of a competition between cipher designers. The winning design, also known as RIJNDAEL, was constructed by Joan Daemen and Vincent Rijnmen.

Like all the AES candidates, the winning design uses a block size of 128 bits, or 16 octets, and variable key-size, 128, 192 and 256 bits (16, 24 and 32 octets) being the allowed key sizes. It does not have any weak keys. Nettle defines AES in '`<nettle/aes.h>`'.

`struct aes_ctx` [Context struct]

`AES_BLOCK_SIZE` [Constant]

The AES block-size, 16

`AES_MIN_KEY_SIZE` [Constant]

`AES_MAX_KEY_SIZE` [Constant]

`AES_KEY_SIZE` [Constant]
Default AES key size, 32

`void aes_set_encrypt_key` (*struct aes_ctx* *`ctx`, *unsigned* `length`, *const uint8_t* *`key`) [Function]
`void aes_set_decrypt_key` (*struct aes_ctx* *`ctx`, *unsigned* `length`, *const uint8_t* *`key`) [Function]
Initialize the cipher, for encryption or decryption, respectively.

`void aes_encrypt` (*struct aes_ctx* *`ctx`, *unsigned* `length`, *const uint8_t* *`dst`, *uint8_t* *`src`) [Function]
Encryption function. *length* must be an integral multiple of the block size. If it is more than one block, the data is processed in ECB mode. `src` and `dst` may be equal, but they must not overlap in any other way.

`void aes_decrypt` (*struct aes_ctx* *`ctx`, *unsigned* `length`, *const uint8_t* *`dst`, *uint8_t* *`src`) [Function]
Analogous to `aes_encrypt`

6.2.2 ARCFOUR

ARCFOUR is a stream cipher, also known under the trade marked name RC4, and it is one of the fastest ciphers around. A problem is that the key setup of ARCFOUR is quite weak, you should never use keys with structure, keys that are ordinary passwords, or sequences of keys like "secret:1", "secret:2", ... If you have keys that don't look like random bit strings, and you want to use ARCFOUR, always hash the key before feeding it to ARCFOUR. Furthermore, the initial bytes of the generated key stream leak information about the key; for this reason, it is recommended to discard the first 512 bytes of the key stream.

```
/* A more robust key setup function for ARCFOUR */
void
arcfour_set_key_hashed(struct arcfour_ctx *ctx,
                       unsigned length, const uint8_t *key)
{
  struct sha256_ctx hash;
  uint8_t digest[SHA256_DIGEST_SIZE];
  uint8_t buffer[0x200];

  sha256_init(&hash);
  sha256_update(&hash, length, key);
  sha256_digest(&hash, SHA256_DIGEST_SIZE, digest);

  arcfour_set_key(ctx, SHA256_DIGEST_SIZE, digest);
  arcfour_crypt(ctx, sizeof(buffer), buffer, buffer);
}
```

Nettle defines ARCFOUR in '`<nettle/arcfour.h>`'.

`struct arcfour_ctx` [Context struct]

`ARCFOUR_MIN_KEY_SIZE` [Constant]
Minimum key size, 1

`ARCFOUR_MAX_KEY_SIZE` [Constant]
Maximum key size, 256

`ARCFOUR_KEY_SIZE` [Constant]
Default ARCFOUR key size, 16

`void arcfour_set_key` (*struct arcfour_ctx* ***ctx***, *unsigned* ***length***, *const uint8_t* ***key***) [Function]
Initialize the cipher. The same function is used for both encryption and decryption.

`void arcfour_crypt` (*struct arcfour_ctx* ***ctx***, *unsigned* ***length***, *const uint8_t* ***dst***, *uint8_t* ***src***) [Function]
Encrypt some data. The same function is used for both encryption and decryption. Unlike the block ciphers, this function modifies the context, so you can split the data into arbitrary chunks and encrypt them one after another. The result is the same as if you had called `arcfour_crypt` only once with all the data.

6.2.3 ARCTWO

ARCTWO (also known as the trade marked name RC2) is a block cipher specified in RFC 2268. Nettle also include a variation of the ARCTWO set key operation that lack one step, to be compatible with the reverse engineered RC2 cipher description, as described in a Usenet post to `sci.crypt` by Peter Gutmann.

ARCTWO uses a block size of 64 bits, and variable key-size ranging from 1 to 128 octets. Besides the key, ARCTWO also has a second parameter to key setup, the number of effective key bits, `ekb`. This parameter can be used to artificially reduce the key size. In practice, `ekb` is usually set equal to the input key size. Nettle defines ARCTWO in '`<nettle/arctwo.h>`'.

We do not recommend the use of ARCTWO; the Nettle implementation is provided primarily for interoperability with existing applications and standards.

`struct arctwo_ctx` [Context struct]

`ARCTWO_BLOCK_SIZE` [Constant]
The AES block-size, 8

`ARCTWO_MIN_KEY_SIZE` [Constant]

`ARCTWO_MAX_KEY_SIZE` [Constant]

`ARCTWO_KEY_SIZE` [Constant]
Default ARCTWO key size, 8

`void arctwo_set_key_ekb` (*struct arctwo_ctx* ***ctx***, *unsigned* ***length***, *const uint8_t* ***key***, *unsigned* ***ekb***) [Function]

`void arctwo_set_key` (*struct arctwo_ctx* ***ctx***, *unsigned* ***length***, *const uint8_t* ***key***) [Function]

`void arctwo_set_key_gutmann` (*struct arctwo_ctx* ***ctx***, *unsigned* ***length***, *const uint8_t* ***key***) [Function]

Initialize the cipher. The same function is used for both encryption and decryption. The first function is the most general one, which lets you provide both the variable size key, and the desired effective key size (in bits). The maximum value for *ekb* is 1024, and for convenience, `ekb = 0` has the same effect as `ekb = 1024`.

`arctwo_set_key(ctx, length, key)` is equivalent to `arctwo_set_key_ekb(ctx, length, key, 8*length)`, and `arctwo_set_key_gutmann(ctx, length, key)` is equivalent to `arctwo_set_key_ekb(ctx, length, key, 1024)`

`void arctwo_encrypt` (*struct arctwo_ctx* ***ctx***, *unsigned* ***length***, *const uint8_t* ***dst***, *uint8_t* ***src***) [Function]

Encryption function. *length* must be an integral multiple of the block size. If it is more than one block, the data is processed in ECB mode. `src` and `dst` may be equal, but they must not overlap in any other way.

`void arctwo_decrypt` (*struct arctwo_ctx* ***ctx***, *unsigned* ***length***, *const uint8_t* ***dst***, *uint8_t* ***src***) [Function]

Analogous to `arctwo_encrypt`

6.2.4 CAST128

CAST-128 is a block cipher, specified in *RFC 2144*. It uses a 64 bit (8 octets) block size, and a variable key size of up to 128 bits. Nettle defines cast128 in '`<nettle/cast128.h>`'.

`struct cast128_ctx` [Context struct]

`CAST128_BLOCK_SIZE` [Constant]

The CAST128 block-size, 8

`CAST128_MIN_KEY_SIZE` [Constant]

Minimum CAST128 key size, 5

`CAST128_MAX_KEY_SIZE` [Constant]

Maximum CAST128 key size, 16

`CAST128_KEY_SIZE` [Constant]

Default CAST128 key size, 16

`void cast128_set_key` (*struct cast128_ctx* ***ctx***, *unsigned* ***length***, *const uint8_t* ***key***) [Function]

Initialize the cipher. The same function is used for both encryption and decryption.

`void cast128_encrypt` (*struct cast128_ctx* ***ctx***, *unsigned* ***length***, *const uint8_t* ***dst***, *uint8_t* ***src***) [Function]

Encryption function. *length* must be an integral multiple of the block size. If it is more than one block, the data is processed in ECB mode. `src` and `dst` may be equal, but they must not overlap in any other way.

`void cast128_decrypt` (*struct cast128_ctx* ***ctx***, *unsigned* ***length***, *const uint8_t* ***dst***, *uint8_t* ***src***) [Function]

Analogous to `cast128_encrypt`

6.2.5 BLOWFISH

BLOWFISH is a block cipher designed by Bruce Schneier. It uses a block size of 64 bits (8 octets), and a variable key size, up to 448 bits. It has some weak keys. Nettle defines BLOWFISH in '`<nettle/blowfish.h>`'.

`struct blowfish_ctx` [Context struct]

`BLOWFISH_BLOCK_SIZE` [Constant]
The BLOWFISH block-size, 8

`BLOWFISH_MIN_KEY_SIZE` [Constant]
Minimum BLOWFISH key size, 8

`BLOWFISH_MAX_KEY_SIZE` [Constant]
Maximum BLOWFISH key size, 56

`BLOWFISH_KEY_SIZE` [Constant]
Default BLOWFISH key size, 16

`int blowfish_set_key` (*struct blowfish_ctx* ***ctx**, *unsigned* **length**, *const uint8_t* ***key**) [Function]
Initialize the cipher. The same function is used for both encryption and decryption. Returns 1 on success, and 0 if the key was weak. Calling `blowfish_encrypt` or `blowfish_decrypt` with a weak key will crash with an assert violation.

`void blowfish_encrypt` (*struct blowfish_ctx* ***ctx**, *unsigned* **length**, *const uint8_t* ***dst**, *uint8_t* ***src**) [Function]
Encryption function. *length* must be an integral multiple of the block size. If it is more than one block, the data is processed in ECB mode. `src` and `dst` may be equal, but they must not overlap in any other way.

`void blowfish_decrypt` (*struct blowfish_ctx* ***ctx**, *unsigned* **length**, *const uint8_t* ***dst**, *uint8_t* ***src**) [Function]
Analogous to `blowfish_encrypt`

6.2.6 DES

DES is the old Data Encryption Standard, specified by NIST. It uses a block size of 64 bits (8 octets), and a key size of 56 bits. However, the key bits are distributed over 8 octets, where the least significant bit of each octet is used for parity. A common way to use DES is to generate 8 random octets in some way, then set the least significant bit of each octet to get odd parity, and initialize DES with the resulting key.

The key size of DES is so small that keys can be found by brute force, using specialized hardware or lots of ordinary work stations in parallel. One shouldn't be using plain DES at all today, if one uses DES at all one should be using "triple DES", see DES3 below.

DES also has some weak keys. Nettle defines DES in '`<nettle/des.h>`'.

`struct des_ctx` [Context struct]

`DES_BLOCK_SIZE` [Constant]
The DES block-size, 8

`DES_KEY_SIZE` [Constant]
DES key size, 8

`int des_set_key` (*struct des_ctx* ***ctx***, *const uint8_t* ***key***) [Function]
Initialize the cipher. The same function is used for both encryption and decryption. Returns 1 on success, and 0 if the key was weak or had bad parity. Calling `des_encrypt` or `des_decrypt` with a bad key will crash with an assert violation.

`void des_encrypt` (*struct des_ctx* ***ctx***, *unsigned* ***length***, *const uint8_t* ***dst***, *uint8_t* ***src***) [Function]
Encryption function. *length* must be an integral multiple of the block size. If it is more than one block, the data is processed in ECB mode. `src` and `dst` may be equal, but they must not overlap in any other way.

`void des_decrypt` (*struct des_ctx* ***ctx***, *unsigned* ***length***, *const uint8_t* ***dst***, *uint8_t* ***src***) [Function]
Analogous to `des_encrypt`

`void des_fix_parity` (*unsigned* ***length***, *uint8_t* ***dst***, *const uint8_t* ***src***) [Function]
Adjusts the parity bits to match DES's requirements. You need this function if you have created a random-looking string by a key agreement protocol, and want to use it as a DES key. *dst* and *src* may be equal.

6.2.7 DES3

The inadequate key size of DES has already been mentioned. One way to increase the key size is to pipe together several DES boxes with independent keys. It turns out that using two DES ciphers is not as secure as one might think, even if the key size of the combination is a respectable 112 bits.

The standard way to increase DES's key size is to use three DES boxes. The mode of operation is a little peculiar: the middle DES box is wired in the reverse direction. To encrypt a block with DES3, you encrypt it using the first 56 bits of the key, then *decrypt* it using the middle 56 bits of the key, and finally encrypt it again using the last 56 bits of the key. This is known as "ede" triple-DES, for "encrypt-decrypt-encrypt".

The "ede" construction provides some backward compatibility, as you get plain single DES simply by feeding the same key to all three boxes. That should help keeping down the gate count, and the price, of hardware circuits implementing both plain DES and DES3.

DES3 has a key size of 168 bits, but just like plain DES, useless parity bits are inserted, so that keys are represented as 24 octets (192 bits). As a 112 bit key is large enough to make brute force attacks impractical, some applications uses a "two-key" variant of triple-DES. In this mode, the same key bits are used for the first and the last DES box in the pipe, while the middle box is keyed independently. The two-key variant is believed to be secure, i.e. there are no known attacks significantly better than brute force.

Naturally, it's simple to implement triple-DES on top of Nettle's DES functions. Nettle includes an implementation of three-key "ede" triple-DES, it is defined in the same place as plain DES, '`<nettle/des.h>`'.

`struct des3_ctx` [Context struct]

`DES3_BLOCK_SIZE` [Constant]
The DES3 block-size is the same as DES_BLOCK_SIZE, 8

`DES3_KEY_SIZE` [Constant]
DES key size, 24

`int des3_set_key` (*struct des3_ctx* ***ctx**, *const uint8_t* ***key**) [Function]
Initialize the cipher. The same function is used for both encryption and decryption. Returns 1 on success, and 0 if the key was weak or had bad parity. Calling `des_encrypt` or `des_decrypt` with a bad key will crash with an assert violation.

For random-looking strings, you can use `des_fix_parity` to adjust the parity bits before calling `des3_set_key`.

`void des3_encrypt` (*struct des3_ctx* ***ctx**, *unsigned* **length**, *const uint8_t* ***dst**, *uint8_t* ***src**) [Function]
Encryption function. *length* must be an integral multiple of the block size. If it is more than one block, the data is processed in ECB mode. `src` and `dst` may be equal, but they must not overlap in any other way.

`void des3_decrypt` (*struct des3_ctx* ***ctx**, *unsigned* **length**, *const uint8_t* ***dst**, *uint8_t* ***src**) [Function]
Analogous to `des_encrypt`

6.2.8 SERPENT

SERPENT is one of the AES finalists, designed by Ross Anderson, Eli Biham and Lars Knudsen. Thus, the interface and properties are similar to AES'. One peculiarity is that it is quite pointless to use it with anything but the maximum key size, smaller keys are just padded to larger ones. Nettle defines SERPENT in '`<nettle/serpent.h>`'.

`struct serpent_ctx` [Context struct]

`SERPENT_BLOCK_SIZE` [Constant]
The SERPENT block-size, 16

`SERPENT_MIN_KEY_SIZE` [Constant]
Minimum SERPENT key size, 16

`SERPENT_MAX_KEY_SIZE` [Constant]
Maximum SERPENT key size, 32

`SERPENT_KEY_SIZE` [Constant]
Default SERPENT key size, 32

`void serpent_set_key` (*struct serpent_ctx* ***ctx**, *unsigned* **length**, *const uint8_t* ***key**) [Function]
Initialize the cipher. The same function is used for both encryption and decryption.

`void serpent_encrypt` (*struct serpent_ctx* ***ctx***, *unsigned* ***length***, *const uint8_t* ***dst***, *uint8_t* ***src***) [Function]

Encryption function. *length* must be an integral multiple of the block size. If it is more than one block, the data is processed in ECB mode. `src` and `dst` may be equal, but they must not overlap in any other way.

`void serpent_decrypt` (*struct serpent_ctx* ***ctx***, *unsigned* ***length***, *const uint8_t* ***dst***, *uint8_t* ***src***) [Function]

Analogous to `serpent_encrypt`

6.2.9 TWOFISH

Another AES finalist, this one designed by Bruce Schneier and others. Nettle defines it in '`<nettle/twofish.h>`'.

`struct twofish_ctx` [Context struct]

`TWOFISH_BLOCK_SIZE` [Constant]

The TWOFISH block-size, 16

`TWOFISH_MIN_KEY_SIZE` [Constant]

Minimum TWOFISH key size, 16

`TWOFISH_MAX_KEY_SIZE` [Constant]

Maximum TWOFISH key size, 32

`TWOFISH_KEY_SIZE` [Constant]

Default TWOFISH key size, 32

`void twofish_set_key` (*struct twofish_ctx* ***ctx***, *unsigned* ***length***, *const uint8_t* ***key***) [Function]

Initialize the cipher. The same function is used for both encryption and decryption.

`void twofish_encrypt` (*struct twofish_ctx* ***ctx***, *unsigned* ***length***, *const uint8_t* ***dst***, *uint8_t* ***src***) [Function]

Encryption function. *length* must be an integral multiple of the block size. If it is more than one block, the data is processed in ECB mode. `src` and `dst` may be equal, but they must not overlap in any other way.

`void twofish_decrypt` (*struct twofish_ctx* ***ctx***, *unsigned* ***length***, *const uint8_t* ***dst***, *uint8_t* ***src***) [Function]

Analogous to `twofish_encrypt`

6.2.10 `struct nettle_cipher`

Nettle includes a struct including information about some of the more regular cipher functions. It should be considered a little experimental, but can be useful for applications that need a simple way to handle various algorithms. Nettle defines these structs in '`<nettle/nettle-meta.h>`'.

`struct nettle_cipher` *name context_size block_size key_size set_encrypt_key set_decrypt_key encrypt decrypt* [Meta struct]

The last four attributes are function pointers, of types `nettle_set_key_func` and `nettle_crypt_func`. The first argument to these functions is a `void *` pointer to a context struct, which is of size `context_size`.

`struct nettle_cipher nettle_aes128` [Constant Struct]
`struct nettle_cipher nettle_aes192` [Constant Struct]
`struct nettle_cipher nettle_aes256` [Constant Struct]
`struct nettle_cipher nettle_arcfour128` [Constant Struct]
`struct nettle_cipher nettle_cast128` [Constant Struct]
`struct nettle_cipher nettle_serpent128` [Constant Struct]
`struct nettle_cipher nettle_serpent192` [Constant Struct]
`struct nettle_cipher nettle_serpent256` [Constant Struct]
`struct nettle_cipher nettle_twofish128` [Constant Struct]
`struct nettle_cipher nettle_twofish192` [Constant Struct]
`struct nettle_cipher nettle_twofish256` [Constant Struct]
`struct nettle_cipher nettle_arctwo40;` [Constant Struct]
`struct nettle_cipher nettle_arctwo64;` [Constant Struct]
`struct nettle_cipher nettle_arctwo128;` [Constant Struct]
`struct nettle_cipher nettle_arctwo_gutmann128;` [Constant Struct]

Nettle includes such structs for all the *regular* ciphers, i.e. ones without weak keys or other oddities.

6.3 Cipher modes

Cipher modes of operation specifies the procedure to use when encrypting a message that is larger than the cipher's block size. As explained in See Section 6.2 [Cipher functions], page 11, splitting the message into blocks and processing them independently with the block cipher (Electronic Code Book mode, ECB) leaks information. Besides ECB, Nettle provides two other modes of operation: Cipher Block Chaining (CBC) and Counter mode (CTR). CBC is widely used, but there are a few subtle issues of information leakage. CTR was standardized more recently, and is believed to be more secure.

6.3.1 Cipher Block Chaining

When using CBC mode, plaintext blocks are not encrypted independently of each other, like in Electronic Cook Book mode. Instead, when encrypting a block in CBC mode, the previous ciphertext block is XORed with the plaintext before it is fed to the block cipher. When encrypting the first block, a random block called an *IV*, or Initialization Vector, is used as the "previous ciphertext block". The IV should be chosen randomly, but it need not be kept secret, and can even be transmitted in the clear together with the encrypted data.

In symbols, if `E_k` is the encryption function of a block cipher, and `IV` is the initialization vector, then `n` plaintext blocks `M_1`,... `M_n` are transformed into `n` ciphertext blocks `C_1`,... `C_n` as follows:

```
C_1 = E_k(IV  XOR M_1)
C_2 = E_k(C_1 XOR M_2)
```

```
...

C_n = E_k(C_(n-1) XOR M_n)
```

Nettle's includes two functions for applying a block cipher in Cipher Block Chaining (CBC) mode, one for encryption and one for decryption. These functions uses `void *` to pass cipher contexts around.

`void cbc_encrypt (`*void *`**`ctx`*, *nettle_crypt_func* **`f`**, *unsigned* **`block_size`**, *uint8_t *`**`iv`*, *unsigned* **`length`**, *uint8_t *`**`dst`*, *const uint8_t *`**`src`**`)` [Function]

`void cbc_decrypt (`*void *`**`ctx`*, *void* (*`*`**`f`**`)*(), *unsigned* **`block_size`**, *uint8_t *`**`iv`*, *unsigned* **`length`**, *uint8_t *`**`dst`*, *const uint8_t *`**`src`**`)` [Function]

Applies the encryption or decryption function *f* in CBC mode. The final ciphertext block processed is copied into *iv* before returning, so that large message be processed be a sequence of calls to `cbc_encrypt`. The function *f* is of type

`void f (void *`*`ctx`*`, unsigned` *`length`*`, uint8_t` *`dst`*`, const uint8_t *`*`src`*`)`,

and the `cbc_encrypt` and `cbc_decrypt` functions pass their argument *ctx* on to *f*.

There are also some macros to help use these functions correctly.

`CBC_CTX (`*`context_type`*, *`block_size`*`)` [Macro]

Expands into

```
{
   context_type ctx;
   uint8_t iv[block_size];
}
```

It can be used to define a CBC context struct, either directly,

```
struct CBC_CTX(struct aes_ctx, AES_BLOCK_SIZE) ctx;
```

or to give it a struct tag,

```
struct aes_cbc_ctx CBC_CTX (struct aes_ctx, AES_BLOCK_SIZE);
```

`CBC_SET_IV (`*`ctx`*, *`iv`*`)` [Macro]

First argument is a pointer to a context struct as defined by `CBC_CTX`, and the second is a pointer to an Initialization Vector (IV) that is copied into that context.

`CBC_ENCRYPT (`*`ctx`*, *`f`*, *`length`*, *`dst`*, *`src`*`)` [Macro]

`CBC_DECRYPT (`*`ctx`*, *`f`*, *`length`*, *`dst`*, *`src`*`)` [Macro]

A simpler way to invoke `cbc_encrypt` and `cbc_decrypt`. The first argument is a pointer to a context struct as defined by `CBC_CTX`, and the second argument is an encryption or decryption function following Nettle's conventions. The last three arguments define the source and destination area for the operation.

These macros use some tricks to make the compiler display a warning if the types of *f* and *ctx* don't match, e.g. if you try to use an `struct aes_ctx` context with the `des_encrypt` function.

6.3.2 Counter mode

Counter mode (CTR) uses the block cipher as a keyed pseudo-random generator. The output of the generator is XORed with the data to be encrypted. It can be understood as a way to transform a block cipher to a stream cipher.

The message is divided into `n` blocks `M_1`,... `M_n`, where `M_n` is of size `m` which may be smaller than the block size. Except for the last block, all the message blocks must be of size equal to the cipher's block size.

If `E_k` is the encryption function of a block cipher, `IC` is the initial counter, then the `n` plaintext blocks are transformed into `n` ciphertext blocks `C_1`,... `C_n` as follows:

```
C_1 = E_k(IC) XOR M_1
C_2 = E_k(IC + 1) XOR M_2

...

C_(n-1) = E_k(IC + n - 2) XOR M_(n-1)
C_n = E_k(IC + n - 1) [1..m] XOR M_n
```

The IC is the initial value for the counter, it plays a similar role as the IV for CBC. When adding, `IC + x`, IC is interpreted as an integer, in network byte order. For the last block, `E_k(IC + n - 1) [1..m]` means that the cipher output is truncated to `m` bytes.

`void ctr_crypt` (*void *ctx*, *nettle_crypt_func* ***f***, *unsigned* ***block_size***, *uint8_t *ctr*, *unsigned* ***length***, *uint8_t *dst*, *const uint8_t *src*) [Function]

Applies the encryption function *f* in CTR mode. Note that for CTR mode, encryption and decryption is the same operation, and hence *f* should always be the encryption function for the underlying block cipher.

When a message is encrypted using a sequence of calls to `ctr_crypt`, all but the last call *must* use a length that is a multiple of the block size.

Like for CBC, there are also a couple of helper macros.

`CTR_CTX` (*context_type*, *block_size*) [Macro]

Expands into

```
{
  context_type ctx;
  uint8_t ctr[block_size];
}
```

`CTR_SET_COUNTER` (*ctx*, *iv*) [Macro]

First argument is a pointer to a context struct as defined by `CTR_CTX`, and the second is a pointer to an initial counter that is copied into that context.

`CTR_CRYPT` (*ctx*, *f*, *length*, *dst*, *src*) [Macro]

A simpler way to invoke `ctr_crypt`. The first argument is a pointer to a context struct as defined by `CTR_CTX`, and the second argument is an encryption function following Nettle's conventions. The last three arguments define the source and destination area for the operation.

6.4 Keyed Hash Functions

A *keyed hash function*, or *Message Authentication Code* (MAC) is a function that takes a key and a message, and produces fixed size MAC. It should be hard to compute a message and a matching MAC without knowledge of the key. It should also be hard to compute the key given only messages and corresponding MACs.

Keyed hash functions are useful primarily for message authentication, when Alice and Bob shares a secret: The sender, Alice, computes the MAC and attaches it to the message. The receiver, Bob, also computes the MAC of the message, using the same key, and compares that to Alice's value. If they match, Bob can be assured that the message has not been modified on its way from Alice.

However, unlike digital signatures, this assurance is not transferable. Bob can't show the message and the MAC to a third party and prove that Alice sent that message. Not even if he gives away the key to the third party. The reason is that the *same* key is used on both sides, and anyone knowing the key can create a correct MAC for any message. If Bob believes that only he and Alice knows the key, and he knows that he didn't attach a MAC to a particular message, he knows it must be Alice who did it. However, the third party can't distinguish between a MAC created by Alice and one created by Bob.

Keyed hash functions are typically a lot faster than digital signatures as well.

6.4.1 HMAC

One can build keyed hash functions from ordinary hash functions. Older constructions simply concatenate secret key and message and hashes that, but such constructions have weaknesses. A better construction is HMAC, described in *RFC 2104*.

For an underlying hash function `H`, with digest size `l` and internal block size `b`, HMAC-H is constructed as follows: From a given key `k`, two distinct subkeys `k_i` and `k_o` are constructed, both of length `b`. The HMAC-H of a message `m` is then computed as `H(k_o | H(k_i | m))`, where `|` denotes string concatenation.

HMAC keys can be of any length, but it is recommended to use keys of length `l`, the digest size of the underlying hash function `H`. Keys that are longer than `b` are shortened to length `l` by hashing with `H`, so arbitrarily long keys aren't very useful.

Nettle's HMAC functions are defined in '`<nettle/hmac.h>`'. There are abstract functions that use a pointer to a `struct nettle_hash` to represent the underlying hash function and `void *` pointers that point to three different context structs for that hash function. There are also concrete functions for HMAC-MD5, HMAC-SHA1, and HMAC-SHA256. First, the abstract functions:

`void hmac_set_key` (*void* ***outer***, *void* ***inner***, *void* ***state***, *const struct nettle_hash* ***H***, *unsigned* ***length***, *const uint8_t* ***key***) [Function]

Initializes the three context structs from the key. The *outer* and *inner* contexts corresponds to the subkeys `k_o` and `k_i`. *state* is used for hashing the message, and is initialized as a copy of the *inner* context.

`void hmac_update` (*void* ***state***, *const struct nettle_hash* ***H***, *unsigned* ***length***, *const uint8_t* ***data***) [Function]

This function is called zero or more times to process the message. Actually, `hmac_update(state, H, length, data)` is equivalent to `H->update(state,`

`length, data)`, so if you wish you can use the ordinary update function of the underlying hash function instead.

`void hmac_digest` (*const void *`outer`, const void *`inner`, void *`state`, const struct nettle_hash *`H`, unsigned `length`, uint8_t *`digest`*) [Function]
Extracts the MAC of the message, writing it to *digest*. *outer* and *inner* are not modified. *length* is usually equal to `H->digest_size`, but if you provide a smaller value, only the first *length* octets of the MAC are written.

This function also resets the *state* context so that you can start over processing a new message (with the same key).

Like for CBC, there are some macros to help use these functions correctly.

`HMAC_CTX` (*`type`*) [Macro]
Expands into

```
{
   type outer;
   type inner;
   type state;
}
```

It can be used to define a HMAC context struct, either directly,

```
struct HMAC_CTX(struct md5_ctx) ctx;
```

or to give it a struct tag,

```
struct hmac_md5_ctx HMAC_CTX (struct md5_ctx);
```

`HMAC_SET_KEY` (*`ctx`, `H`, `length`, `key`*) [Macro]
ctx is a pointer to a context struct as defined by `HMAC_CTX`, *H* is a pointer to a `const struct nettle_hash` describing the underlying hash function (so it must match the type of the components of *ctx*). The last two arguments specify the secret key.

`HMAC_DIGEST` (*`ctx`, `H`, `length`, `digest`*) [Macro]
ctx is a pointer to a context struct as defined by `HMAC_CTX`, *H* is a pointer to a `const struct nettle_hash` describing the underlying hash function. The last two arguments specify where the digest is written.

Note that there is no `HMAC_UPDATE` macro; simply call `hmac_update` function directly, or the update function of the underlying hash function.

6.4.2 Concrete HMAC functions

Now we come to the specialized HMAC functions, which are easier to use than the general HMAC functions.

6.4.2.1 HMAC-MD5

`struct hmac_md5_ctx` [Context struct]

`void hmac_md5_set_key` (*struct hmac_md5_ctx *`ctx`, unsigned `key_length`, const uint8_t *`key`*) [Function]
Initializes the context with the key.

`void hmac_md5_update` (*struct hmac_md5_ctx* ***ctx**, *unsigned* **length**, *const uint8_t* ***data**) [Function]

Process some more data.

`void hmac_md5_digest` (*struct hmac_md5_ctx* ***ctx**, *unsigned* **length**, *uint8_t* ***digest**) [Function]

Extracts the MAC, writing it to *digest*. *length* may be smaller than `MD5_DIGEST_SIZE`, in which case only the first *length* octets of the MAC are written.

This function also resets the context for processing new messages, with the same key.

6.4.2.2 HMAC-SHA1

`struct hmac_sha1_ctx` [Context struct]

`void hmac_sha1_set_key` (*struct hmac_sha1_ctx* ***ctx**, *unsigned* **key_length**, *const uint8_t* ***key**) [Function]

Initializes the context with the key.

`void hmac_sha1_update` (*struct hmac_sha1_ctx* ***ctx**, *unsigned* **length**, *const uint8_t* ***data**) [Function]

Process some more data.

`void hmac_sha1_digest` (*struct hmac_sha1_ctx* ***ctx**, *unsigned* **length**, *uint8_t* ***digest**) [Function]

Extracts the MAC, writing it to *digest*. *length* may be smaller than `SHA1_DIGEST_SIZE`, in which case only the first *length* octets of the MAC are written.

This function also resets the context for processing new messages, with the same key.

6.4.2.3 HMAC-SHA256

`struct hmac_sha256_ctx` [Context struct]

`void hmac_sha256_set_key` (*struct hmac_sha256_ctx* ***ctx**, *unsigned* **key_length**, *const uint8_t* ***key**) [Function]

Initializes the context with the key.

`void hmac_sha256_update` (*struct hmac_sha256_ctx* ***ctx**, *unsigned* **length**, *const uint8_t* ***data**) [Function]

Process some more data.

`void hmac_sha256_digest` (*struct hmac_sha256_ctx* ***ctx**, *unsigned* **length**, *uint8_t* ***digest**) [Function]

Extracts the MAC, writing it to *digest*. *length* may be smaller than `SHA256_DIGEST_SIZE`, in which case only the first *length* octets of the MAC are written.

This function also resets the context for processing new messages, with the same key.

6.5 Public-key algorithms

Nettle uses GMP, the GNU bignum library, for all calculations with large numbers. In order to use the public-key features of Nettle, you must install GMP, at least version 3.0, before compiling Nettle, and you need to link your programs with `-lhogweed -lnettle -lgmp`.

The concept of *Public-key* encryption and digital signatures was discovered by Whitfield Diffie and Martin E. Hellman and described in a paper 1976. In traditional, "symmetric", cryptography, sender and receiver share the same keys, and these keys must be distributed in a secure way. And if there are many users or entities that need to communicate, each *pair* needs a shared secret key known by nobody else.

Public-key cryptography uses trapdoor one-way functions. A *one-way function* is a function `F` such that it is easy to compute the value `F(x)` for any `x`, but given a value `y`, it is hard to compute a corresponding `x` such that `y = F(x)`. Two examples are cryptographic hash functions, and exponentiation in certain groups.

A *trapdoor one-way function* is a function `F` that is one-way, unless one knows some secret information about `F`. If one knows the secret, it is easy to compute both `F` and it's inverse. If this sounds strange, look at the RSA example below.

Two important uses for one-way functions with trapdoors are public-key encryption, and digital signatures. The public-key encryption functions in Nettle are not yet documented; the rest of this chapter is about digital signatures.

To use a digital signature algorithm, one must first create a *key-pair*: A public key and a corresponding private key. The private key is used to sign messages, while the public key is used for verifying that that signatures and messages match. Some care must be taken when distributing the public key; it need not be kept secret, but if a bad guy is able to replace it (in transit, or in some user's list of known public keys), bad things may happen.

There are two operations one can do with the keys. The signature operation takes a message and a private key, and creates a signature for the message. A signature is some string of bits, usually at most a few thousand bits or a few hundred octets. Unlike paper-and-ink signatures, the digital signature depends on the message, so one can't cut it out of context and glue it to a different message.

The verification operation takes a public key, a message, and a string that is claimed to be a signature on the message, and returns true or false. If it returns true, that means that the three input values matched, and the verifier can be sure that someone went through with the signature operation on that very message, and that the "someone" also knows the private key corresponding to the public key.

The desired properties of a digital signature algorithm are as follows: Given the public key and pairs of messages and valid signatures on them, it should be hard to compute the private key, and it should also be hard to create a new message and signature that is accepted by the verification operation.

Besides signing meaningful messages, digital signatures can be used for authorization. A server can be configured with a public key, such that any client that connects to the service is given a random nonce message. If the server gets a reply with a correct signature matching the nonce message and the configured public key, the client is granted access. So the configuration of the server can be understood as "grant access to whoever knows the private key corresponding to this particular public key, and to no others".

6.5.1 RSA

The RSA algorithm was the first practical digital signature algorithm that was constructed. It was described 1978 in a paper by Ronald Rivest, Adi Shamir and L.M. Adleman, and the technique was also patented in the USA in 1983. The patent expired on September 20, 2000, and since that day, RSA can be used freely, even in the USA.

It's remarkably simple to describe the trapdoor function behind RSA. The "one-way"-function used is

```
F(x) = x^e mod n
```

I.e. raise x to the `e`:th power, while discarding all multiples of `n`. The pair of numbers `n` and `e` is the public key. `e` can be quite small, even `e = 3` has been used, although slightly larger numbers are recommended. `n` should be about 1000 bits or larger.

If `n` is large enough, and properly chosen, the inverse of F, the computation of `e`:th roots modulo `n`, is very difficult. But, where's the trapdoor?

Let's first look at how RSA key-pairs are generated. First `n` is chosen as the product of two large prime numbers `p` and `q` of roughly the same size (so if `n` is 1000 bits, `p` and `q` are about 500 bits each). One also computes the number `phi = (p-1)(q-1)`, in mathematical speak, `phi` is the order of the multiplicative group of integers modulo n.

Next, `e` is chosen. It must have no factors in common with `phi` (in particular, it must be odd), but can otherwise be chosen more or less randomly. `e = 65537` is a popular choice, because it makes raising to the `e`:th power particularly efficient, and being prime, it usually has no factors common with `phi`.

Finally, a number `d`, `d < n` is computed such that `e d mod phi = 1`. It can be shown that such a number exists (this is why `e` and `phi` must have no common factors), and that for all x,

```
(x^e)^d mod n = x^(ed) mod n = (x^d)^e mod n = x
```

Using Euclid's algorithm, `d` can be computed quite easily from `phi` and `e`. But it is still hard to get `d` without knowing `phi`, which depends on the factorization of `n`.

So `d` is the trapdoor, if we know `d` and `y = F(x)`, we can recover x as `y^d mod n`. `d` is also the private half of the RSA key-pair.

The most common signature operation for RSA is defined in *PKCS#1*, a specification by RSA Laboratories. The message to be signed is first hashed using a cryptographic hash function, e.g. MD5 or SHA1. Next, some padding, the ASN.1 "Algorithm Identifier" for the hash function, and the message digest itself, are concatenated and converted to a number `x`. The signature is computed from `x` and the private key as `s = x^d mod n`[1]. The signature, `s` is a number of about the same size of `n`, and it usually encoded as a sequence of octets, most significant octet first.

The verification operation is straight-forward, `x` is computed from the message in the same way as above. Then `s^e mod n` is computed, the operation returns true if and only if the result equals `x`.

[1] Actually, the computation is not done like this, it is done more efficiently using `p`, `q` and the Chinese remainder theorem (CRT). But the result is the same.

6.5.2 Nettle's RSA support

Nettle represents RSA keys using two structures that contain large numbers (of type `mpz_t`).

`rsa_public_key` *size n e* [Context struct]
: `size` is the size, in octets, of the modulo, and is used internally. `n` and `e` is the public key.

`rsa_private_key` *size d p q a b c* [Context struct]
: `size` is the size, in octets, of the modulo, and is used internally. `d` is the secret exponent, but it is not actually used when signing. Instead, the factors `p` and `q`, and the parameters `a`, `b` and `c` are used. They are computed from `p`, `q` and `e` such that `a e mod (p - 1) = 1, b e mod (q - 1) = 1, c q mod p = 1`.

Before use, these structs must be initialized by calling one of

`void rsa_public_key_init (`*struct rsa_public_key* ***pub**`)` [Function]
`void rsa_private_key_init (`*struct rsa_private_key* ***key**`)` [Function]
: Calls `mpz_init` on all numbers in the key struct.

and when finished with them, the space for the numbers must be deallocated by calling one of

`void rsa_public_key_clear (`*struct rsa_public_key* ***pub**`)` [Function]
`void rsa_private_key_clear (`*struct rsa_private_key* ***key**`)` [Function]
: Calls `mpz_clear` on all numbers in the key struct.

In general, Nettle's RSA functions deviates from Nettle's "no memory allocation"-policy. Space for all the numbers, both in the key structs above, and temporaries, are allocated dynamically. For information on how to customize allocation, see See Section "GMP Allocation" in *GMP Manual*.

When you have assigned values to the attributes of a key, you must call

`int rsa_public_key_prepare (`*struct rsa_public_key* ***pub**`)` [Function]
`int rsa_private_key_prepare (`*struct rsa_private_key* ***key**`)` [Function]
: Computes the octet size of the key (stored in the `size` attribute, and may also do other basic sanity checks. Returns one if successful, or zero if the key can't be used, for instance if the modulo is smaller than the minimum size specified by PKCS#1.

Before signing or verifying a message, you first hash it with the appropriate hash function. You pass the hash function's context struct to the RSA signature function, and it will extract the message digest and do the rest of the work. There are also alternative functions that take the MD5 or SHA1 hash digest as argument.

Creation and verification of signatures is done with the following functions:

`void rsa_md5_sign (`*const struct rsa_private_key* ***key**, *struct md5_ctx* ***hash**, *mpz_t* ***signature**`)` [Function]
`void rsa_sha1_sign (`*const struct rsa_private_key* ***key**, *struct sha1_ctx* ***hash**, *mpz_t* ***signature**`)` [Function]

`void rsa_sha256_sign` (*const struct rsa_private_key* ***key***, *struct sha256_ctx* ***hash***, *mpz_t* ***signature***) [Function]

The signature is stored in *signature* (which must have been `mpz_init`:ed earlier). The hash context is reset so that it can be used for new messages.

`void rsa_md5_sign_digest` (*const struct rsa_private_key* ***key***, *const uint8_t* ***digest***, *mpz_t* ***signature***) [Function]

`void rsa_sha1_sign_digest` (*const struct rsa_private_key* ***key***, *const uint8_t* ***digest***, *mpz_t* ***signature***); [Function]

`void rsa_sha256_sign_digest` (*const struct rsa_private_key* ***key***, *const uint8_t* ***digest***, *mpz_t* ***signature***); [Function]

Creates a signature from the given hash digest. *digest* should point to a digest of size `MD5_DIGEST_SIZE`, `SHA1_DIGEST_SIZE`, or `SHA256_DIGEST_SIZE`, respectively. The signature is stored in *signature* (which must have been `mpz_init`:ed earlier)

`int rsa_md5_verify` (*const struct rsa_public_key* ***key***, *struct md5_ctx* ***hash***, *const mpz_t* ***signature***) [Function]

`int rsa_sha1_verify` (*const struct rsa_public_key* ***key***, *struct sha1_ctx* ***hash***, *const mpz_t* ***signature***) [Function]

`int rsa_sha256_verify` (*const struct rsa_public_key* ***key***, *struct sha256_ctx* ***hash***, *const mpz_t* ***signature***) [Function]

Returns 1 if the signature is valid, or 0 if it isn't. In either case, the hash context is reset so that it can be used for new messages.

`int rsa_md5_verify_digest` (*const struct rsa_public_key* ***key***, *const uint8_t* ***digest***, *const mpz_t* ***signature***) [Function]

`int rsa_sha1_verify_digest` (*const struct rsa_public_key* ***key***, *const uint8_t* ***digest***, *const mpz_t* ***signature***) [Function]

`int rsa_sha256_verify_digest` (*const struct rsa_public_key* ***key***, *const uint8_t* ***digest***, *const mpz_t* ***signature***) [Function]

Returns 1 if the signature is valid, or 0 if it isn't. *digest* should point to a digest of size `MD5_DIGEST_SIZE`, `SHA1_DIGEST_SIZE`, or `SHA256_DIGEST_SIZE`, respectively.

If you need to use the RSA trapdoor, the private key, in a way that isn't supported by the above functions Nettle also includes a function that computes `x^d mod n` and nothing more, using the CRT optimization.

`void rsa_compute_root` (*struct rsa_private_key* ***key***, *mpz_t* ***x***, *const mpz_t* ***m***) [Function]

Computes `x = m^d`, efficiently.

At last, how do you create new keys?

`int rsa_generate_keypair` (*struct rsa_public_key* ***pub***, *struct rsa_private_key* ***key***, *void* ***random_ctx***, *nettle_random_func* ***random***, *void* ***progress_ctx***, *nettle_progress_func* ***progress***, *unsigned* ***n_size***, *unsigned* ***e_size***); [Function]

There are lots of parameters. *pub* and *key* is where the resulting key pair is stored. The structs should be initialized, but you don't need to call `rsa_public_key_prepare` or `rsa_private_key_prepare` after key generation.

random_ctx and *random* is a randomness generator. `random(random_ctx, length, dst)` should generate `length` random octets and store them at `dst`. For advice, see See Section 6.6 [Randomness], page 33.

progress and *progress_ctx* can be used to get callbacks during the key generation process, in order to uphold an illusion of progress. *progress* can be NULL, in that case there are no callbacks.

size_n is the desired size of the modulo, in bits. If *size_e* is non-zero, it is the desired size of the public exponent and a random exponent of that size is selected. But if *e_size* is zero, it is assumed that the caller has already chosen a value for `e`, and stored it in *pub*. Returns 1 on success, and 0 on failure. The function can fail for example if if *n_size* is too small, or if *e_size* is zero and `pub->e` is an even number.

6.5.3 Nettle's DSA support

The DSA digital signature algorithm is more complex than RSA. It was specified during the early 1990s, and in 1994 NIST published FIPS 186 which is the authoritative specification. Sometimes DSA is referred to using the acronym DSS, for Digital Signature Standard.

For DSA, the underlying mathematical problem is the computation of discreet logarithms. The public key consists of a large prime `p`, a small prime `q` which is a factor of `p-1`, a number `g` which generates a subgroup of order `q` modulo `p`, and an element `y` in that subgroup.

The size of `q` is fixed to 160 bits, to match with the SHA1 hash algorithm which is used in DSA. The size of `q` is in principle unlimited, but the standard specifies only nine specific sizes: `512 + l*64`, where `l` is between 0 and 8. Thus, the maximum size of `p` is 1024 bits, at that is also the recommended size.

The subgroup requirement means that if you compute

```
g^t mod p
```

for all possible integers `t`, you will get precisely `q` distinct values.

The private key is a secret exponent `x`, such that

```
g^x = y mod p
```

In mathematical speak, `x` is the *discrete logarithm* of `y` mod `p`, with respect to the generator `d`. The size of `x` will also be about 160 bits.

The signature generation algorithm is randomized; in order to create a DSA signature, you need a good source for random numbers (see Section 6.6 [Randomness], page 33).

To create a signature, one starts with the hash digest of the message, `h`, which is a 160 bit number, and a random number `k, 0<k<q`, also 160 bits. Next, one computes

```
r = (g^k mod p) mod q
s = k^-1 (h + x r) mod q
```

The signature is the pair `(r, s)`, two 160 bit numbers. Note the two different mod operations when computing `r`, and the use of the secret exponent `x`.

To verify a signature, one first checks that `0 < r,s < q`, and then one computes backwards,

```
w = s^-1 mod q
v = (g^(w h) y^(w r) mod p) mod q
```

The signature is valid if `v = r`. This works out because `w = s^-1 mod q = k (h + x r)^-1 mod q`, so that

```
g^(w h) y^(w r) = g^(w h) (g^x)^(w r) = g^(w (h + x r)) = g^k
```

When reducing mod `q` this yields `r`. Note that when verifying a signature, we don't know either `k` or `x`: those numbers are secret.

If you can choose between RSA and DSA, which one is best? Both are believed to be secure. DSA gained popularity in the late 1990s, as a patent free alternative to RSA. Now that the RSA patents have expired, there's no compelling reason to want to use DSA.

DSA signatures are smaller than RSA signatures, which is important for some specialized applications.

From a practical point of view, DSA's need for a good randomness source is a serious disadvantage. If you ever use the same `k` (and `r`) for two different message, you leak your private key.

6.5.4 Nettle's DSA support

Like for RSA, Nettle represents DSA keys using two structures, containing values of type `mpz_t`. For information on how to customize allocation, see See Section "GMP Allocation" in *GMP Manual*.

Most of the DSA functions are very similar to the corresponding RSA functions, but there are a few differences pointed out below. For a start, there are no functions corresponding to `rsa_public_key_prepare` and `rsa_private_key_prepare`.

`dsa_public_key` *p q g y* [Context struct]
: The public parameters described above.

`dsa_private_key` *x* [Context struct]
: The private key `x`.

Before use, these structs must be initialized by calling one of

`void dsa_public_key_init` (*struct dsa_public_key ***pub***) [Function]
`void dsa_private_key_init` (*struct dsa_private_key ***key***) [Function]
: Calls `mpz_init` on all numbers in the key struct.

When finished with them, the space for the numbers must be deallocated by calling one of

`void dsa_public_key_clear` (*struct dsa_public_key ***pub***) [Function]
`void dsa_private_key_clear` (*struct dsa_private_key ***key***) [Function]
: Calls `mpz_clear` on all numbers in the key struct.

Signatures are represented using the structure below, and need to be initialized and cleared in the same way as the key structs.

`dsa_signature` *r s* [Context struct]

`void dsa_signature_init` (*struct dsa_signature ***signature***) [Function]
`void dsa_signature_clear` (*struct dsa_signature ***signature***) [Function]
: You must call `dsa_signature_init` before creating or using a signature, and call `dsa_signature_clear` when you are finished with it.

For signing, you need to provide both the public and the private key (unlike RSA, where the private key struct includes all information needed for signing), and a source for random numbers. Signatures always use the SHA1 hash function.

`void dsa_sign` (*const struct dsa_public_key* ***pub***, *const struct dsa_private_key* ***key***, *void* ***random_ctx***, *nettle_random_func* ***random***, *struct sha1_ctx* ***hash***, *struct dsa_signature* ***signature***) [Function]

`void dsa_sign_digest` (*const struct dsa_public_key* ***pub***, *const struct dsa_private_key* ***key***, *void* ***random_ctx***, *nettle_random_func* ***random***, *const uint8_t* ***digest***, *struct dsa_signature* ***signature***) [Function]

Creates a signature from the given hash context or digest. *random_ctx* and *random* is a randomness generator. `random(random_ctx, length, dst)` should generate `length` random octets and store them at `dst`. For advice, see See Section 6.6 [Randomness], page 33.

Verifying signatures is a little easier, since no randomness generator is needed. The functions are

`int dsa_verify` (*const struct dsa_public_key* ***key***, *struct sha1_ctx* ***hash***, *const struct dsa_signature* ***signature***) [Function]

`int dsa_verify_digest` (*const struct dsa_public_key* ***key***, *const uint8_t* ***digest***, *const struct dsa_signature* ***signature***) [Function]

Verifies a signature. Returns 1 if the signature is valid, otherwise 0.

Key generation uses mostly the same parameters as the corresponding RSA function.

`int dsa_generate_keypair` (*struct dsa_public_key* ***pub***, *struct dsa_private_key* ***key***, *void* ***random_ctx***, *nettle_random_func* ***random***, *void* ***progress_ctx***, *nettle_progress_func* ***progress***, *unsigned* ***bits***) [Function]

pub and *key* is where the resulting key pair is stored. The structs should be initialized before you call this function.

random_ctx and *random* is a randomness generator. `random(random_ctx, length, dst)` should generate `length` random octets and store them at `dst`. For advice, see See Section 6.6 [Randomness], page 33.

progress and *progress_ctx* can be used to get callbacks during the key generation process, in order to uphold an illusion of progress. *progress* can be NULL, in that case there are no callbacks.

bits is the desired size of `p`, in bits. To generate keys that conform to the standard, you must use a value of the form `512 + l*64`, for `0 <= l <= 8`. Keys smaller than 768 bits are not considered secure, so you should probably stick to 1024. Non-standard sizes are possible, in particular sizes larger than 1024 bits, although DSA implementations can not in general be expected to support such keys. Also note that using very large keys doesn't make much sense, because the security is also limited by the size of the smaller prime `q`, which is always 160 bits.

Returns 1 on success, and 0 on failure. The function will fail if *bits* is too small.

6.6 Randomness

A crucial ingredient in many cryptographic contexts is randomness: Let `p` be a random prime, choose a random initialization vector `iv`, a random key `k` and a random exponent `e`, etc. In the theories, it is assumed that you have plenty of randomness around. If this assumption is not true in practice, systems that are otherwise perfectly secure, can be broken. Randomness has often turned out to be the weakest link in the chain.

In non-cryptographic applications, such as games as well as scientific simulation, a good randomness generator usually means a generator that has good statistical properties, and is seeded by some simple function of things like the current time, process id, and host name.

However, such a generator is inadequate for cryptography, for at least two reasons:

- It's too easy for an attacker to guess the initial seed. Even if it will take some 2^32 tries before he guesses right, that's far too easy. For example, if the process id is 16 bits, the resolution of "current time" is one second, and the attacker knows what day the generator was seeded, there are only about 2^32 possibilities to try if all possible values for the process id and time-of-day are tried.
- The generator output reveals too much. By observing only a small segment of the generator's output, its internal state can be recovered, and from there, all previous output and all future output can be computed by the attacker.

A randomness generator that is used for cryptographic purposes must have better properties. Let's first look at the seeding, as the issues here are mostly independent of the rest of the generator. The initial state of the generator (its seed) must be unguessable by the attacker. So what's unguessable? It depends on what the attacker already knows. The concept used in information theory to reason about such things is called "entropy", or "conditional entropy" (not to be confused with the thermodynamic concept with the same name). A reasonable requirement is that the seed contains a conditional entropy of at least some 80-100 bits. This property can be explained as follows: Allow the attacker to ask `n` yes-no-questions, of his own choice, about the seed. If the attacker, using this question-and-answer session, as well as any other information he knows about the seeding process, still can't guess the seed correctly, then the conditional entropy is more than `n` bits.

Let's look at an example. Say information about timing of received network packets is used in the seeding process. If there is some random network traffic going on, this will contribute some bits of entropy or "unguessability" to the seed. However, if the attacker can listen in to the local network, or if all but a small number of the packets were transmitted by machines that the attacker can monitor, this additional information makes the seed easier for the attacker to figure out. Even if the information is exactly the same, the conditional entropy, or unguessability, is smaller for an attacker that knows some of it already before the hypothetical question-and-answer session.

Seeding of good generators is usually based on several sources. The key point here is that the amount of unguessability that each source contributes, depends on who the attacker is. Some sources that have been used are:

High resolution timing of i/o activities
: Such as completed blocks from spinning hard disks, network packets, etc. Getting access to such information is quite system dependent, and not all systems include suitable hardware. If available, it's one of the better randomness source one can find in a digital, mostly predictable, computer.

User activity
: Timing and contents of user interaction events is another popular source that is available for interactive programs (even if I suspect that it is sometimes used in order to make the user feel good, not because the quality of the input is needed or used properly). Obviously, not available when a machine is unattended. Also beware of networks: User interaction that happens across a long serial cable, TELNET session, or even SSH session may be visible to an attacker, in full or partially.

Audio input
: Any room, or even a microphone input that's left unconnected, is a source of some random background noise, which can be fed into the seeding process.

Specialized hardware
: Hardware devices with the sole purpose of generating random data have been designed. They range from radioactive samples with an attached Geiger counter, to amplification of the inherent noise in electronic components such as diodes and resistors, to low-frequency sampling of chaotic systems. Hashing successive images of a Lava lamp is a spectacular example of the latter type.

Secret information
: Secret information, such as user passwords or keys, or private files stored on disk, can provide some unguessability. A problem is that if the information is revealed at a later time, the unguessability vanishes. Another problem is that this kind of information tends to be fairly constant, so if you rely on it and seed your generator regularly, you risk constructing almost similar seeds or even constructing the same seed more than once.

For all practical sources, it's difficult but important to provide a reliable lower bound on the amount of unguessability that it provides. Two important points are to make sure that the attacker can't observe your sources (so if you like the Lava lamp idea, remember that you have to get your own lamp, and not put it by a window or anywhere else where strangers can see it), and that hardware failures are detected. What if the bulb in the Lava lamp, which you keep locked into a cupboard following the above advice, breaks after a few months?

So let's assume that we have been able to find an unguessable seed, which contains at least 80 bits of conditional entropy, relative to all attackers that we care about (typically, we must at the very least assume that no attacker has root privileges on our machine).

How do we generate output from this seed, and how much can we get? Some generators (notably the Linux '`/dev/random`' generator) tries to estimate available entropy and restrict the amount of output. The goal is that if you read 128 bits from '`/dev/random`', you should get 128 "truly random" bits. This is a property that is useful in some specialized circumstances, for instance when generating key material for a one time pad, or when working with unconditional blinding, but in most cases, it doesn't matter much. For most application, there's no limit on the amount of useful "random" data that we can generate from a small seed; what matters is that the seed is unguessable and that the generator has good cryptographic properties.

At the heart of all generators lies its internal state. Future output is determined by the internal state alone. Let's call it the generator's key. The key is initialized from the unguessable seed. Important properties of a generator are:

Key-hiding
: An attacker observing the output should not be able to recover the generator's key.

Independence of outputs
: Observing some of the output should not help the attacker to guess previous or future output.

Forward secrecy
: Even if an attacker compromises the generator's key, he should not be able to guess the generator output *before* the key compromise.

Recovery from key compromise
: If an attacker compromises the generator's key, he can compute *all* future output. This is inevitable if the generator is seeded only once, at startup. However, the generator can provide a reseeding mechanism, to achieve recovery from key compromise. More precisely: If the attacker compromises the key at a particular time `t_1`, there is another later time `t_2`, such that if the attacker observes all output generated between `t_1` and `t_2`, he still can't guess what output is generated after `t_2`.

Nettle includes one randomness generator that is believed to have all the above properties, and two simpler ones.

ARCFOUR, like any stream cipher, can be used as a randomness generator. Its output should be of reasonable quality, if the seed is hashed properly before it is used with `arcfour_set_key`. There's no single natural way to reseed it, but if you need reseeding, you should be using Yarrow instead.

The "lagged Fibonacci" generator in '`<nettle/knuth-lfib.h>`' is a fast generator with good statistical properties, but is **not** for cryptographic use, and therefore not documented here. It is included mostly because the Nettle test suite needs to generate some test data from a small seed.

The recommended generator to use is Yarrow, described below.

6.6.1 Yarrow

Yarrow is a family of pseudo-randomness generators, designed for cryptographic use, by John Kelsey, Bruce Schneier and Niels Ferguson. Yarrow-160 is described in a paper at `http://www.counterpane.com/yarrow.html`, and it uses SHA1 and triple-DES, and has a 160-bit internal state. Nettle implements Yarrow-256, which is similar, but uses SHA256 and AES to get an internal state of 256 bits.

Yarrow was an almost finished project, the paper mentioned above is the closest thing to a specification for it, but some smaller details are left out. There is no official reference implementation or test cases. This section includes an overview of Yarrow, but for the details of Yarrow-256, as implemented by Nettle, you have to consult the source code. Maybe a complete specification can be written later.

Yarrow can use many sources (at least two are needed for proper reseeding), and two randomness "pools", referred to as the "slow pool" and the "fast pool". Input from the sources is fed alternatingly into the two pools. When one of the sources has contributed 100 bits of entropy to the fast pool, a "fast reseed" happens and the fast pool is mixed into the internal state. When at least two of the sources have contributed at least 160 bits each to the slow pool, a "slow reseed" takes place. The contents of both pools are mixed into the internal state. These procedures should ensure that the generator will eventually recover after a key compromise.

The output is generated by using AES to encrypt a counter, using the generator's current key. After each request for output, another 256 bits are generated which replace the key. This ensures forward secrecy.

Yarrow can also use a *seed file* to save state across restarts. Yarrow is seeded by either feeding it the contents of the previous seed file, or feeding it input from its sources until a slow reseed happens.

Nettle defines Yarrow-256 in '`<nettle/yarrow.h>`'.

`struct yarrow256_ctx` [Context struct]

`struct yarrow_source` [Context struct]
Information about a single source.

`YARROW256_SEED_FILE_SIZE` [Constant]
Recommanded size of the Yarrow-256 seed file.

`void yarrow256_init` (*struct yarrow256_ctx* ***ctx**, *unsigned* **nsources**, *struct yarrow_source* ***sources**) [Function]
Initializes the yarrow context, and its *nsources* sources. It's possible to call it with *nsources*=0 and *sources*=NULL, if you don't need the update features.

`void yarrow256_seed` (*struct yarrow256_ctx* ***ctx**, *unsigned* **length**, *uint8_t* ***seed_file**) [Function]
Seeds Yarrow-256 from a previous seed file. *length* should be at least `YARROW256_SEED_FILE_SIZE`, but it can be larger.

The generator will trust you that the *seed_file* data really is unguessable. After calling this function, you *must* overwrite the old seed file with newly generated data from `yarrow256_random`. If it's possible for several processes to read the seed file at about the same time, access must be coordinated using some locking mechanism.

`int yarrow256_update` (*struct yarrow256_ctx* ***ctx**, *unsigned* **source**, *unsigned* **entropy**, *unsigned* **length**, *const uint8_t* ***data**) [Function]
Updates the generator with data from source *SOURCE* (an index that must be smaller than the number of sources). *entropy* is your estimated lower bound for the entropy in the data, measured in bits. Calling update with zero *entropy* is always safe, no matter if the data is random or not.

Returns 1 if a reseed happened, in which case an application using a seed file may want to generate new seed data with `yarrow256_random` and overwrite the seed file. Otherwise, the function returns 0.

`void yarrow256_random` (*struct yarrow256_ctx* ***ctx***, *unsigned* ***length***, *uint8_t* ***dst***) [Function]

Generates *length* octets of output. The generator must be seeded before you call this function.

If you don't need forward secrecy, e.g. if you need non-secret randomness for initialization vectors or padding, you can gain some efficiency by buffering, calling this function for reasonably large blocks of data, say 100-1000 octets at a time.

`int yarrow256_is_seeded` (*struct yarrow256_ctx* ***ctx***) [Function]

Returns 1 if the generator is seeded and ready to generate output, otherwise 0.

`unsigned yarrow256_needed_sources` (*struct yarrow256_ctx* ***ctx***) [Function]

Returns the number of sources that must reach the threshold before a slow reseed will happen. Useful primarily when the generator is unseeded.

`void yarrow256_fast_reseed` (*struct yarrow256_ctx* ***ctx***) [Function]
`void yarrow256_slow_reseed` (*struct yarrow256_ctx* ***ctx***) [Function]

Causes a fast or slow reseed to take place immediately, regardless of the current entropy estimates of the two pools. Use with care.

Nettle includes an entropy estimator for one kind of input source: User keyboard input.

`struct yarrow_key_event_ctx` [Context struct]

Information about recent key events.

`void yarrow_key_event_init` (*struct yarrow_key_event_ctx* ***ctx***) [Function]

Initializes the context.

`unsigned yarrow_key_event_estimate` (*struct yarrow_key_event_ctx* ***ctx***, *unsigned* ***key***, *unsigned* ***time***) [Function]

key is the id of the key (ASCII value, hardware key code, X keysym, ..., it doesn't matter), and *time* is the timestamp of the event. The time must be given in units matching the resolution by which you read the clock. If you read the clock with microsecond precision, *time* should be provided in units of microseconds. But if you use `gettimeofday` on a typical Unix system where the clock ticks 10 or so microseconds at a time, *time* should be given in units of 10 microseconds.

Returns an entropy estimate, in bits, suitable for calling `yarrow256_update`. Usually, 0, 1 or 2 bits.

6.7 Miscellaneous functions

`uint8_t * memxor` (*uint8_t* ***dst***, *const uint8_t* ***src***, *size_t* ***n***) [Function]

XORs the source area on top of the destination area. The interface doesn't follow the Nettle conventions, because it is intended to be similar to the ANSI-C `memcpy` function.

`memxor` is declared in '`<nettle/memxor.h>`'.

6.8 Compatibility functions

For convenience, Nettle includes alternative interfaces to some algorithms, for compatibility with some other popular crypto toolkits. These are not fully documented here; refer to the source or to the documentation for the original implementation.

MD5 is defined in [RFC 1321], which includes a reference implementation. Nettle defines a compatible interface to MD5 in '`<nettle/md5-compat.h>`'. This file defines the typedef `MD5_CTX`, and declares the functions `MD5Init`, `MD5Update` and `MD5Final`.

Eric Young's "libdes" (also part of OpenSSL) is a quite popular DES implementation. Nettle includes a subset if its interface in '`<nettle/des-compat.h>`'. This file defines the typedefs `des_key_schedule` and `des_cblock`, two constants `DES_ENCRYPT` and `DES_DECRYPT`, and declares one global variable `des_check_key`, and the functions `des_cbc_cksum` `des_cbc_encrypt`, `des_ecb2_encrypt`, `des_ecb3_encrypt`, `des_ecb_encrypt`, `des_ede2_cbc_encrypt`, `des_ede3_cbc_encrypt`, `des_is_weak_key`, `des_key_sched`, `des_ncbc_encrypt` `des_set_key`, and `des_set_odd_parity`.

7 Traditional Nettle Soup

For the serious nettle hacker, here is a recipe for nettle soup. 4 servings.

1 liter fresh nettles (urtica dioica)

2 tablespoons butter

3 tablespoons flour

1 liter stock (meat or vegetable)

1/2 teaspoon salt

a tad white pepper

some cream or milk

Gather 1 liter fresh nettles. Use gloves! Small, tender shoots are preferable but the tops of larger nettles can also be used.

Rinse the nettles very well. Boil them for 10 minutes in lightly salted water. Strain the nettles and save the water. Hack the nettles. Melt the butter and mix in the flour. Dilute with stock and the nettle-water you saved earlier. Add the hacked nettles. If you wish you can add some milk or cream at this stage. Bring to a boil and let boil for a few minutes. Season with salt and pepper.

Serve with boiled egg-halves.

8 Installation

Nettle uses `autoconf`. To build it, unpack the source and run

```
./configure
make
make check
make install
```

to install in the default location, '`/usr/local`'. The library files are installed in '`/use/local/lib/libnettle.a`' '`/use/local/lib/libhogweed.a`' and the include files are installed in '`/use/local/include/nettle/`'.

To get a list of configure options, use `./configure --help`.

By default, only static libraries are built and installed. To also build and install shared libraries, use the ' `--enable-shared`' option to `./configure`.

Using GNU make is recommended. For other make programs, in particular BSD make, you may have to use the '`--disable-dependency-tracking`' option to `./configure`.

Function and Concept Index

O

P

R

S

T

Y