HASHLIB

Industry-Standard Cryptography on the TI-84+ CE

Version 9.2

Quick Reference

by Anthony Cagliano

Contents

Installation

`HASHLIB` is a library, not a program or an application. It is meant to be used in accordance with the same usage guidelines as the other libraries in the CE C toolchain. If you have not already, familiarize yourself with the installation and usage instructions for the toolchain starting with this page.

If you are an end user who needs the library present, either for testing or for using a program that requires HASHLIB, install it by sending the library file, the TI application variable `hashlib.8xv` to your device. If you do not do this, programs that use it will fail to start, instead returning to the homescreen and yelling at you about a missing library.

If you are a developer looking to use the library within your own project, you must follow the steps below.

- Move the library's C header and .lib files to the correct directories. Copy the `hashlib.lib` file to the `lib/libload` directory within the extracted toolchain folder. Copy the `hashlib.h` folder to the `include` directory.
- Include the C header for HASHLIB in any C source file where you use anything from the library. Do this like so: `#include <hashlib.h>`.
- Use defines or functions from the library freely within any C source file where the library's header is included.

1 Enumerations, Definitions, and Macros

Hash Algorithms	
Identifier	**Algorithm**
SHA256	Selects the SHA-256 hash or hmac algorithm

AES Cipher Modes	
Identifier	**Description**
AES_MODE_CBC	Selects cyclic-block chaining (CBC) cipher mode
AES_MODE_CTR	Selects counter (CTR) cipher mode

AES Padding Schemes	
Identifier	**Description**
PAD_DEFAULT	Enables default padding mode (PKCS#7)
PAD_PKCS7	Enables the PKCS#7 padding scheme
PAD_ISO2	Enables the ISO-9797 M2 padding scheme

AES Response Codes	
Identifier	**Description**
AES_OK	AES operation completed w/o errors
AES_INVALID_ARG	One or more inputs invalid
AES_INVALID_MSG	Message cannot be encrypted
AES_INVALID_CIPHERMODE	Cipher mode not supported
AES_INVALID_PADDINGMODE	Padding mode not supported
AES_INVALID_CIPHERTEXT	Ciphertext cannot be decrypted

RSA Response Codes	
Identifier	**Description**
RSA_OK	RSA operation completed w/o errors
RSA_INVALID_ARG	One or more inputs invalid
RSA_INVALID_MSG	Message value exceeds modulus value
RSA_INVALID_MODULUS	Modulus not odd Length not in range 128-256 bytes
RSA_ENCODING_ERROR	OAEP requirements not met, ex: Message not less than modulus length minus twice the hash digest length plus two.

Constant Definitions	
Identifier	**Description**
fastRam_Safe	Region of fast RAM generally safe to use for short-term computations
fastRam_Unsafe	Region of fast RAM used by this library for PRNG and hashing speed
SHA256_DIGEST_LEN	Binary length of SHA-256 hash digest (32 bytes)
AES_BLOCKSIZE	Block length of the AES cipher (16 bytes)
AES_IVSIZE	Length of the AES initialization vector (same as block size)

Macros	
Identifier	**Description**
aes_outsize(len)	Returns the smallest multiple of the block size that can hold the ciphertext with any required padding
aes_extoutsize(len)	Returns the output of `aes_outsize(len)` with an additional 16 bytes added for the IV

2 Implementations

2.1 Cryptographically-Secure Random Number Generator (CSRNG)

Many of the pseudo random number generators (PRNGs) you find in computers, even the one within the C toolchain for the CE, are insecure for cryptographic purposes. They produce statistical randomness but they are *deterministic*, meaning that for a single input there is a single output, regardless of the complexity of the algorithm that generates that output. While these types of generators suffice for the *illusion of randomness*, they are insecure for cryptographic purposes. A deterministic RNG can be defeated by an adversary who gains knowledge of the state of the generator, allowing them to compute every output that follows. Furthermore many of these RNGs are *seeded* (def: set to an initial value) using some known (or easily computable/reverse-engineerable) value like system time or CPU cycle counter.

A secure RNG requires some form of entropy gathering. *Entropy* is defined as unpredictability. The more entropy that exists in a source, the more unpredictable its output is likely to be. Modern computer systems derive cryptographically-secure randomness by pooling entropy from various sources: network and bus noise, CPU clock jitter, electrical variances, and more. The calculator is a much simpler device and lacks the vast majority of these sources of entropy. However, even on such a simple device you can produce cryptographically-secure randomness in theory by pooling enough entropy from the one source that exists–bus noise. *Bus noise* is the term for electrical variance across a region of unmapped memory. A more technical explanation of the sourcing of randomness as well as a proof of cryptographic security can be found in *HASHLIB Cryptanalysis, Section 1.*

It is also worth noting that on CEmu the bus noise is simulated using a deterministic RNG. This means that while this SRNG will still be statistically random on that platform, it will not be unpredictable and thus the security proofs do not hold for that platform.

bool csrand_init(void)

Initializes the CSRNG. Returns **True** if the source selection succeeded and **False** if it failed. Be sure to intercept and handle a return value of False from this function.

uint32_t csrand_get(void)

Returns a securely pseudo random 32-bit (4 byte) unsigned integer.

bool csrand_fill(void* buffer, size_t size)

buffer Pointer to an arbitrary buffer to fill with random bytes.
size Number of bytes to write.

2.2 Cryptographic Hashing

A cryptographic hash is a fixed-size representation of an arbitrary-length stream of data. Hashes are also non-invertible, meaning that you cannot return the original message from the hash. The main function of a cryptographic hash is to verify whether some block of stored or transmitted data has changed from its original creation. This works because the deviation of even a single bit in the input changes the hash quite drastically.

Hashes have a number of practical uses, not just in cryptography but throughout the field of information security. A few of these uses are:

- **File integrity monitoring**: A database of hashes for known good files is saved and consistently checked against the current state of a system. Changes to the current hashes can reveal potentially malicious tampering with files.
- **Data transfer integrity**: Including a hash with data sent over the Internet can have a number of benefits. Firstly, if packets are lost between the source and destination, a mismatch between the included hash and one computed by the destination would reveal the transfer as corrupted and then a well-designed transmission control protocol would initiate a re-transmission. A similar benefit is in the detection of malicious tampering of the message in transit. It is not possible to differentiate between packet loss and malicious tampering, and so any message that fails a transfer integrity check should never be accepted.
- **Password encryption**: Hashes are used to encrypt passwords as well. Because hashes are non-invertible, you cannot un-hash a password to check it later, for example, when you supply your password to log in to some account. Generally, when you **create** a password, the password and a salt (random oracle meant to make password resistant to dictionary attacks) are hashed and then the hash and salt are dumped into a database. The next time you supply your password, the salt is retrieved from the database and used with the password you have input and the hash is recomputed. If the output hash matches the hash that is in the database, you have entered the correct password.

void hash_init(hash_ctx* ctx, uint8_t hash_alg)

Initializes the hash-state context for use.
ctx A pointer to an instance of *hash_ctx*.
hash_alg The hashing algorithm to use. See `hash_algorithms` (Enumerations).

void hash_update(hash_ctx* ctx, const void* data, size_t len)

Updates the hash-state with new data. Be sure to initialize it first!
ctx A pointer to an instance of *hash_ctx*.
data A pointer to arbitrary data to hash.
len The size, in bytes, of the data to hash.

void hash_final(hash_ctx* ctx, void* digest)

Performs final transformations on the context and returns a digest from the current hash-state.
Does not destroy the context. It can still be used with the same data stream if needed.
ctx A pointer to an instance of *hash_ctx*.
digest A pointer to a buffer to write the digest to.

2.3 Hash-Based Message Authentication Code (HMAC)

An HMAC generates a more secure hash by using a key known only to authorized parties as part of the hash initialization. Thus, while normal hashes can be generated and verified by anyone, only the parties with the key can generate and validate using a HMAC hash. An HMAC can fill the same roles as a normal cryptographic hash, but provides endpoint validation as well.

void hmac_init(hmac_ctx* ctx, const void* key, size_t keylen, uint8_t hash_alg)

Initializes the HMAC hash-state context for use.
ctx A pointer to an instance of *hmac_ctx*.
key A pointer to the key to use in the HMAC initilaization.
keylen The length of the key, in bytes.
hash_alg The hashing algorithm to use. See `hash_algorithms` (Enumerations).

NIST recommends a minimum key length of 128 bits, or 16 bytes.

void hmac_update(hmac_ctx* ctx, const void* data, size_t len)

Updates the HMAC hash-state with new data. Be sure to initialize it first!
ctx A pointer to an instance of *hmac_ctx*.
data A pointer to arbitrary data to hash.
len The size, in bytes, of the data to hash.

void hmac_final(hmac_ctx* ctx, void* digest)

Performs final transformations on the context and returns a digest from the current hash-state. Does not destroy the context. It can still be used with the same data stream if needed.
ctx A pointer to an instance of *hmac_ctx*.
digest A pointer to a buffer to write the digest to.

2.4 Mask and Key Generation

Sometimes in cryptography you need to generate hashes or keys of an arbitrary size. Two related, but different, functions exist to fill this role. The first of the two is a **mask generation function (MGF)**. A MGF generates a mask of arbitrary length by passing the data with a counter appended to it to a cryptographic primative such as SHA-256. The second of the two is a **password-based key derivation function**. A PBKDF works by using the supplied password as the key for an HMAC and then hashing the salt for the given number of rounds for each block of output.

void hash_mgf1(const void* data, size_t datalen,
void* outbuf, size_t outlen, uint8_t hash_alg)

Generates a mask of a given length from the given data.
data A pointer to data to generate the mask with.
datalen The length, in bytes, of the data.
outbuf A pointer to a buffer to write the mask to.
outlen The number of bytes of the mask to output.
hash_alg The hashing algorithm to use. See `hash_algorithms` (Enumerations).

void hmac_pbkdf2(const char* password, size_t passlen,
void* key, size_t keylen
const void* salt, size_t saltlen,
size_t rounds, uint8_t hash_alg)

Generates a key of given length from a password, salt, and a given number of rounds.
password A pointer to a string containing the password.
passlen The length of the password string.
key A pointer to a buffer to write the key to.
keylen The number of bytes of the key to output.
salt A pointer to a buffer containing pseudo random bytes.
saltlen The length of the salt, in bytes.
rounds The number of times to iterate the HMAC function per block in the output.
hash_alg The hashing algorithm to use. See `hash_algorithms` (Enumerations).

NIST recommends a minimum salt length of 128 bits, or 16 bytes.

2.5 Advanced Encryption Standard (AES)

The **Advanced Encryption Standard (AES)** is a symmetric encryption system and a block cipher. Symmetric encryption means that the same key can be used for both encryption and decryption. A block cipher is a cipher in which the data is operated on in blocks of a fixed size. AES is one of the most secure encryption systems in use today, expected to remain secure even through the advent of quantum computing. It is also fast and more secure than asymmetric encryption for smaller key sizes.

The AES implementation available in this library provides confidentiality only, not integrity. For details on authenticated encryption, see Authenticated Encryption with HASHLIB.

aes_error_t aes_init(const aes_ctx* ctx, const void* key, size_t keylen, const void* iv, uint24_t flags)

Configures an AES context given a key and a series of option flags.
ctx Pointer to an AES cipher configuration context.
key Pointer to a buffer containing the AES key.
keylen The length, in bytes, of the AES key.
iv Pointer to initialization vector, buffer equaling the block size in length containing random bytes.
flags A series of cipher options bitwise-ORd together. Pass 0 to use default options.

cipher options flags:
`cipher modes`: AES_MODE_CBC or AES_MODE_CTR
`CBC padding modes`: PAD_DEFAULT or PAD_PKCS7 or PAD_ISO2
`CTR initialization vector fixed nonce length`: AES_CTR_NONCELEN(len)
`CTR initialization vector counter length`: AES_CTR_COUNTERLEN(len)
Passing 0 for flags is functionally equivalent to passing: `AES_MODE_CBC | PAD_PKCS7`.
Passing `AES_MODE_CTR` for flags with no other options set is functionally equivalent to passing:
`AES_MODE_CTR | AES_CTR_NONCELEN(8) | AES_CTR_COUNTERLEN(8)`

Do not edit the cipher context manually after initialization, you may corrupt the state.
Cipher contexts are stateful and one-directional. Once you use a cipher for encryption or decryption a flag is set preventing it from being used in the other direction. If you need a two-way AES session, initialize two contexts using the same key and assign one the IV for the outgoing stream and the other the IV for the incoming stream.

security considerations:
`data length limit:` Ciphers begin leaking data after the same key is used on a certain amount of data. For AES it is recommended that you cycle your key after encrypting 2^{64} blocks of information.
`counter length:` The size of your counter in CTR mode directly impacts the length of data you can encrypt before the counter cycles, and the cipher begins leaking information. Leaving the default behavior is recommended.

aes_error_t aes_encrypt(const aes_ctx* ctx, const void* plaintext, size_t len, void* ciphertext)

Encrypts the given message using the AES cipher.
ctx A pointer to an AES cipher configured by aes_init().
plaintext A pointer to a buffer containing data to encrypt.
len The length of the data to encrypt.
ciphertext A pointer to a buffer to write the encrypted output to.

> The AES context is stateful. After a call to `aes_encrypt()`, you can pass another message on the same stream without altering or re-initializing the context by simply calling `aes_encrypt()` again.
>
> Once a context is used with aes_encrypt(), attempting to use it with aes_decrypt() will return `AES_INVALID_OPERATION`.
>
> Calls to aes_encrypt() are chainable.
> `aes_encrypt(msg1 + msg2) == aes_encrypt(msg1) + aes_encrypt(msg2)`
> If in CBC mode, this will be true only after any appended padding is removed by the decryptor.
>
> If in CBC mode, padding will be appended automatically by the encryptor. **ciphertext** will therefore need to be large enough to hold the plaintext plus any necessary padding. This is the length of the plaintext rounded up to the next multiple of the block size. If the plaintext already is a multiple of the block size, another block of padding is added.

aes_error_t aes_decrypt(const aes_ctx* ctx, const void* ciphertext, size_t len, void* plaintext)

Decrypts the given message using the AES cipher.
ctx A pointer to an AES cipher configured by aes_init().
ciphertext A pointer to a buffer containing data to decrypt.
len The length of the data to decrypt.
plaintext A pointer to a buffer to write the decrypted output to.

> The AES context is stateful. After a call to `aes_decrypt()`, you can pass another message on the same stream without altering or re-initializing the context by simply calling `aes_decrypt()` again.
>
> Once a context is used with aes_decrypt(), attempting to use it with aes_encrypt() will return `AES_INVALID_OPERATION`.
>
> Calls to aes_decrypt() are chainable.
> `aes_decrypt(msg1 + msg2) == aes_decrypt(msg1) + aes_decrypt(msg2)`
> If in CBC mode, this will be true only after any appended padding is removed.
>
> If in CBC mode, padding will not be removed automatically. This can be done algorithmically quite simply by the user. If using PKCS7, simply read the last byte of the padded plaintext and strip that many bytes. If using ISO2, simply seek from the end of the padded plaintext backwards to the first $0x80 byte and strip from that byte forwards.

2.6 RSA Public Key Encryption

Public key encryption is a form of asymmetric encryption generally used to share a secret key for AES or another symmetric encryption system. To communicate between two parties, both need a public key and a private key. The public key is (hence the term ”public”) common knowledge and is sent to other parties in the clear. The private key is known only to the host. The public key is used to encrypt messages for the host, and the private key is used by the host to decrypt those messages. The public key and private key are inverses of each other such that:
$encrypted\ =\ message^{public\ exponent}\ \%\ public\ modulus$
$message\ =\ encrypted^{private\ exponent}\ \%\ private\ modulus$
RSA is very slow, especially on the TI-84+ CE. Encrypting with just a 1024-bit modulus will take several seconds. For this reason, do not use RSA for sustained encrypted communication. Use RSA once to share a key with a remote host, then use AES. Also the RSA implementation in this library is encryption only. This means you will need to handshake with a server to create a secure session, like so:

(a) Connect to remote host. Let that server generate a public and private key pair. Send the public key to the calculator.

(b) Use hashlib to generate an AES secret. Encrypt that secret using RSA and send the encrypted message to the remote host.

(c) Decrypt the message on the server, and set up an AES session using the secret just shared with the remote host.

**rsa_error_t rsa_encrypt(const void* msg, size_t msglen, void* ciphertext,
const void* pubkey, size_t keylen, uint8_t oaep_hash_alg)**

Encrypts the given message using the given public key and the public exponent 65537.
Applies the OAEP v2.2 encoding scheme prior to encryption.
msg A pointer to a buffer containing data to encrypt.
msglen The length of the data to encrypt.
ciphertext A pointer to a buffer to write the encrypted output to.
pubkey A pointer to an RSA public modulus.
keylen The length of the RSA public modulus, in bytes.
oaep_hash_alg The hashing algorithm to use for OAEP. See `hash_algorithms` (Enumerations).

2.7 Miscellaneous Functions

void digest_tostring(const void* digest, size_t len, const char* hexstr)

Outputs a textual representation of the hex encoding of a binary digest.
Ex: 0xfe, 0xa4, 0xc1, 0xf2 => "FEA4C1F2"
digest A pointer to a digest to convert to a string.
len The length of the digest, in bytes, to convert.
hexstr A pointer to a buffer to write the string. Must be equal to twice the digest length + 1.

void digest_compare(const void* digest1, const void* digest2, size_t len)

Compares the given number of bytes at *digest*1 with *digest*2 in a manner that is resistant to timing analysis.
digest1 A pointer to the first buffer to compare.
digest2 A pointer to the second buffer to compare.
len The number of bytes to compare.

2.8 Addendum: Authenticated Encryption with HASHLIB

Authenticated encryption is an encryption scheme that produces a ciphertext that is not only obfuscated but also has its integrity and authenticity verifiable. This can be accomplished in a few ways, the most common of which are: (1) appending a signature, hash, or keyed hash to a message, and (2) implementing a cipher mode that integrates authentication.

#2 above is not implemented in HASHLIB. Most of the authenticating cipher modes are computationally-intensive without hardware acceleration and may not be feasible for use on the TI-84+ CE. While consideration is being given to potentially adding a cipher mode such as OCB or GCM if a sufficiently-optimized implementation for this platform can be found (or devised), it is possible to construct a ciphertext guarded against tampering by using method #1, which this library does provide for.

It is my recommendation that whenever you are sending data you need to be **truly secure** with this library, you always embed a **keyed hash** into the message that the recipient can validate. This functionality is provided by the `HMAC` implementation shown earlier in this document. Proper application of HMAC for ciphertext integrity requires the following considerations:

(a) Initialization vector/nonce blocks for encryption are securely pseudo-random.

(b) Encryption and HMAC keys are also securely pseudo-random and are long enough to be considered secure. Minimum key sizes recommended are 16 bytes.

(c) You are not using your encryption key as your HMAC key or vice versa. There are attack vectors that result from using the same key for encryption and authentication.

(d) Append a keyed hash (HMAC) of the initialization vector/nonce, encrypted message, and any other associated data such as packet headers to the outgoing message. On the receiving side, validate the HMAC before decryption and reject any message that does not authenticate. The HMAC key can be an application secret known to both parties or a generated nonce shared alongside the AES encryption key using RSA or another public key encryption method.

Sample authenticated encryption construction using HASHLIB API

```
// this assumes that the AES secret ‘aes_key‘ and the HMAC secret ‘hmac_key‘
// have been negotiated beforehand.

// let’s send a simple ascii string
char* msg = "The daring fox jumped over the moon."

// the header is a size word, containing size of string plus our IV
// header can really be whatever you want, but some arbitrary nonsense as an example
```

```
    size_t header = sizeof(msg)+AES_IVSIZE;

    uint8_t iv[AES_IVSIZE];
    uint8_t hmac_digest[SHA256_DIGEST_LEN];
    aes_ctx ctx;
    hmac_ctx hmac;

    // !!!! NEVER PROCEED IF csrand_init() FAILS !!!
    if(!csrand_init()) return false;
    csrand_fill(iv, AES_IVSIZE;

    // initialize AES context with mode, key, and iv
    aes_init(&ctx, aes_key, sizeof aes_key, iv, AES_MODE_CTR);

    // encrypt message
    // aes_encrypt supports in-place encryption
    aes_encrypt(&ctx, msg, strlen(msg), msg);

    // hash everything you are sending, except the hash itself
    hmac_init(&hmac, hmac_key, sizeof hmac_key, SHA256);
    hmac_update(&hmac, &header, sizeof header);
    hmac_update(&hmac, iv, sizeof iv);
    hmac_update(&hmac, msg, strlen(msg));
    hmac_final(&hmac, hmac_digest);

    // ps_send is a pseudo-function implying sending a packet segment over network
    ps_send(&header, sizeof header);
    ps_send(iv, sizeof iv);
    ps_send(msg, sizeof msg);
    ps_send(hmac_digest, sizeof hmac_digest);
```

3 Contributors

- Anthony Cagliano [cryptographer, lead developer]
- beckadamtheinventor [contributing developer, assembly conversions]
- commandblockguy [contributing developer]
- Zeroko [information on entropy on TI-84+ CE]
- jacobly [ez80 implementation of digest_compare and _powmod for RSA]

4 Disclaimer

HASHLIB is a work-in-progress and has seen very little time as the forerunning cryptography library for the TI-84+ CE calculator. This means that it has not had much time to be thoroughly analyzed, and due to some hardware constraints may never offer total security against every possible attack. For this reason, I heavily advise that however secure HASHLIB may be, you never use it for encrypting truly sensitive data like online banking and other accounts, credit card information, and the like over an insecure network. It is likely safe enough to be used to encrypt data transfers and account login for TI-84+ CE game servers and package managers like the ones currently under development. By using this software you release and save harmless its developer(s) from liability for data compromise that may arise should you use this software.