HASHLIB

Industry-Standard Cryptography on the TI-84+ CE

Quick Reference

by Anthony Cagliano

Contents

1 Algorithm State Contexts

Context	Contents	Purpose
sha256_ctx	struct { uint8_t data[64]; uint8_t datalen; uint8_t bitlen[8]; uint32_t state[8]; }	Defines the hash-state data for an instance of SHA-256.
hmac_ctx	struct { uint8_t ipad[64]; uint8_t opad[64]; sha256_ctx ctx; }	Defines the hash-state data for an instance of SHA-256 HMAC.
aes_ctx	struct { uint24_t keysize; uint32_t round_keys[60]; }	Defines an AES key schedule context.

2 Defines

Define	Purpose
hashlib_FastMemBufferSafe	Defines a region of fast memory generally safe to use for short-term computations.
hashlib_FastMemBufferUnsafe	Defines a region of fast memory that generally gets used by the SPRNG and SHA-256 algorithms.
SHA256_DIGEST_LEN	Defines the binary-length of a SHA-256 hash digest.
SHA256_HEXDIGEST_LEN	Defines the length of a string derived from a SHA-256 hash digest.
AES_BLOCKSIZE	Defines the block size of the of the AES cipher, in bytes.
AES_IVSIZE	Defines the size of the AES initialization vector.
AES128_KEYLEN	Defines the byte-length of a 128-bit AES key.
AES192_KEYLEN	Defines the byte-length of a 192-bit AES key.
AES256_KEYLEN	Defines the byte-length of a 256-bit AES key.

3 Enumerations

Enumeration	Values	Meaning
aes_cipher_modes	AES_MODE_CBC	AES CBC cipher mode
	AES_MODE_CTR	AES CTR cipher mode
aes_padding_schemes	SCHM_PKCS7	PKCS#7 padding mode
	SCHM_ISO2	ISO-9797 M2 padding mode
	SCHM_DEFAULT	
aes_error_t	AES_OK	AES operation succeeded
	AES_INVALID_ARG	one or more invalid inputs
	AES_INVALID_MSG	message invalid
	AES_INVALID_CIPHERMODE	unrecognized cipher mode
	AES_INVALID_PADDINGMODE	unrecognized padding mode
	AES_INVALID_CIPHERTEXT	invalid ciphertext or bad hash
ssl_sig_modes	SSLSIG_RSA_SHA256	signature algorithm RSA with SHA-256
	SSLSIG_ECDSA	signature algorithm ECDSA (unimpl.)
rsa_error_t	RSA_OK	RSA operation succeeded
	RSA_INVALID_ARG	one or more invalid inputs
	RSA_INVALID_MSG	message too long
	RSA_INVALID_MODULUS	modulus not odd
	RSA_ENCODING_ERROR	OAEP encoding error

4 Macros

Macro	Function
hashlib_AESCiphertextSize(len)	Outputs the required buffer size for a properly padded ciphertext given an input unpadded plaintext length **len**.
hashlib_AESCiphertextIVSize(len)	Outputs the required buffer size for the above with an additional block for an initialization vector.
hashlib_AESKeygen(key, keylen)	An alias of hashlib_RandomBytes(). Generates a psuedorandom AES key of given length.
hashlib_MallocContext(size)	An alias of malloc(). Dynamically allocates space for a context of given size.

5 Implementations

5.1 Secure Psuedorandom Number Generator (SPRNG)

Many of the psuedorandom number generators (PRNGs) you find in computers and even the one within the C toolchain for the CE are insecure for cryptographic purposes. They produce statistical randomness, but the state is generally seeded using a value such as rtc_Time(). If an adversary reconstructs the seed, every output of the PRNG becomes computable with little effort. These types of PRNGs are called deterministic algorithms–given the input, the output is predictable. These PRNGs work for games and other applications where the illusion of randomness is sufficient, but they are not safe for cryptography.

A secure PRNG is a random number generator that is not only statistically random, but also passes the next-bit and state compromise tests. The next-bit test is defined like so:

> Given the prior output of the PRNG (bits 0 -> i), the next bit (i+1) of the output cannot be predicted by a polynomial-time statistical test with a probability non-negligibly greater than 50%.

In simpler terms, a secure PRNG must be unpredictable, or "entropic". The state compromise test is defined like so:

> An adversary gaining knowledge of the initial state of the PRNG does not gain any information about its output.

The PRNG previded by HASHLIB solves both tests like so:
(next-bit)

- The PRNG's output is derived from a 119-byte entropy pool created by reading data from the most entropic byte located within floating memory on the device.
- The "entropic byte" is a byte containing a bit that that, out of 1024 test reads, has the closest to a 50/50 split between 1's and 0's.
- The byte containing that bit is read in xor mode seven times per byte to offset any hardware-based correlation between subsequent reads from the entropic byte.
- The entropy pool is then run through a cryptographic hash to spread that entropy evenly through the returned random value.
- The PRNG produces 96.51 bits of entropy per 32 bit number returned.
- **Assertion**: *A source of randomness with sufficient entropy passed through a cryptographic hash will produce output that passes all statistical randomness tests as well as the next-bit test.*

(state compromise)

- The entropy pool is discarded after it is used once, and a new pool is generated for the next random number. This means that the prior state has no bearing on the next output of the PRNG.
- The PRNG destroys its own state after the random number is generated so that the state used to generate it does not persist in memory.

void hashlib_SPRNGInit(void)

Initializes the SPRNG by polling the 512 bytes from 0xD65800 to 0xD66000 looking for a floating bit within unmapped memory with the least bias.

uint32_t hashlib_SPRNGRandom(void)

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

bool hashlib_RandomBytes(void* buffer, size_t size)

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

5.2 SHA-256 Cryptographic Hash

A cryptographic hash is used to validate that data is unchanged between two endpoints. It is similar to a checksum, but checksums can be easily fooled; cryptographic hashes are a lot harder to fool due to how they distribute the bits in a data stream. The general use of a hash is as follows:

(a) The party sending a message hashes it and includes that hash as part of the message.

(b) The recipient hashes the message (excluding the hash) themselves and then compares that hash to the one included in the message.

(c) If the hashes match, the message is complete and unaltered.

(d) If the hashes don't match, the message is incomplete or has been tampered with and should be discarded.

(e) A SHA-256 HMAC can be used in place of a standard hash to make it more difficult for an attacker to masquerade as the legitimate origin.

void hashlib_Sha256Init(sha256_ctx* ctx)

Initializes the SHA-256 hash-state context for use.
ctx A pointer to an instance of *sha256_ctx*.

void hashlib_Sha256Update(sha256_ctx* ctx, const void* data, size_t len)

Updates the SHA-256 hash-state with new data.
ctx A pointer to an instance of *sha256_ctx*.
data A pointer to arbitrary data to hash.
len The size, in bytes, of the data to hash.

void hashlib_Sha256Final(sha256_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 *sha256_ctx*.
digest A pointer to a buffer to write the digest to.

5.3 SHA-256 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 SHA-256 can be generated and verified by anyone, only the parties with the key can generate and validate using a HMAC hash.

void hashlib_HMACSha256Init(hmac_ctx* ctx, const void* key, size_t keylen)

Initializes the SHA-256 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.

void hashlib_HMACSha256Update(hmac_ctx* ctx, const void* data, size_t len)

Updates the SHA-256 HMAC hash-state with new data.
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 hashlib_HMACSha256Final(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.

void hashlib_HMACSha256Reset(hmac_ctx* ctx)

Resets the SHA-256 HMAC context to its state after a call to hashlib_HMACSha256Init().
ctx A pointer to an instance of *hmac_ctx*.

5.4 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 SHA-256 HMAC and then hashing the salt for the given number of rounds for each block of output.

void hashlib_MGF1Hash(const void* data, size_t datalen, void* outbuf, size_t outlen)

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.

void hashlib_PBKDF2(const char* password, size_t passlen, void* key,
const void* salt, size_t saltlen,
size_t rounds, size_t keylen)

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.
salt A pointer to a buffer containing psuedorandom bytes.
saltlen The length of the salt. Standards recommend at least 128 bits (16 bytes).
rounds The number of times to iterate the SHA-256 HMAC function per block in the output.
keylen The number of bytes of the key to output.

5.5 Advanced Encryption Standard (AES)

The **Advanced Encryption Standard (AES)** is a symmetric encryption system (the same key works for encryption and decryption) and a block cipher (the encryption occurs in blocks of a certain size [16 bytes for AES]). 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.

void hashlib_AESLoadKey(const void* key, const aes_ctx* ks, size_t keylen)

Populates an AES key schedule context given an AES key.
key Pointer to a buffer containing the AES key.
ks Pointer to an AES key schedule to output.
keylen The length, in bytes, of the AES key.

void hashlib_AESEncrypt(const void* plaintext, size_t len, void* ciphertext,
const aes_ctx* ks, const void* iv,
uint8_t ciphermode, uint8_t paddingmode)

Encrypts the given message using the AES cipher, also applying the appropriate padding to the message.
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. Size of this buffer should the smallest multiple of the blocksize that can hold the encrypted data and any necessary padding. See the macros provided to simplify computing this.
ks A pointer to an AES key schedule output by hashlib_AESLoadKey().
iv A pointer to an initialization vector, a buffer equal to the block size in length containing psuedorandom bytes.
ciphermode The cipher mode to use for encryption. See enumerations.
paddingmode The padding mode to use prior to encryption, if needed. See enumerations.

void hashlib_AESDecrypt(const void* ciphertext, size_t len, void* plaintext, const aes_ctx* ks, const void* iv, uint8_t ciphermode, uint8_t paddingmode)

Decrypts the given message using the AES cipher, also stripping the padding from the from message where needed.
ciphertext A pointer to a buffer containing data to decrypt.
len The length of the data to decrypt.
ciphertext A pointer to a buffer to write the decrypted output to.
ks A pointer to an AES key schedule output by hashlib_AESLoadKey().
iv A pointer to an initialization vector, a buffer equal to the block size in length containing psuedo-random bytes. Should be the same as used to encrypt this message.
ciphermode The cipher mode to use for decryption. See enumerations.
paddingmode The padding mode to use after decryption, if needed. See enumerations.

5.6 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.

void hashlib_RSAEncrypt(const void* msg, size_t msglen, void* ciphertext, const void* pubkey, size_t keylen)

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.

void hashlib_SSLVerifySignature(const void* ca_pubkey, size_t keysize, const void* cert, size_t certlen, uint8_t sig_alg)

Validates the signature of the given SSL certificate using the given public key of a certifying authority. Requires a certificate with an RSA w/ SHA-256 signature algorithm.
ca_pubkey A pointer to a buffer containing the public key of the certifying authority.
keysize The length of the public key, in bytes.
cert A pointer to a buffer containing the certificate to validate.
certlen The length of the certificate, in bytes.
sig_alg The signature algorithm to use. As of now only *SSLSIG_RSA_SHA256* is supported.

5.7 Miscellaneous Functions

void hashlib_DigestToHexStr(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 hashlib_EraseContext(void* ctx, size_t len)

Erases the arbitrary buffer given.
ctx A pointer to a context or buffer to erase.
len The number of bytes to erase.

void hashlib_CompareDigest(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.

6 Contributors

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

7 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.